COM and Automation Support for XLISP-STAT on Windows

Luke Tierney [*]

January 21, 2000

Contents

[*]Research supported in part by grant DMS-9971814 from the National Science Foundation.

1 Introduction

This report presents an interface to the Microsoft Component Object Model (COM), in particular the Automation subset of COM. All functions and variables are contained in the `WIN32-COM` (nickname `COM`) package. The next section presents an outline of the interface and some examples. The implementation is given in the following sections.

Eventually this interface will be folded into the standard distribution. For now, it is made available as a zip file, `win32com.zip`[1]. This file should be unpacked in the `Autoload` subdirectory of the Xlisp-Stat tree. This will insure that the interface is loaded when it is used. Be sure to preserve the directory structure when unpacking the zip file. You will also need the Win32 support library[2].

If you want to install the COM server, you need to run the following command after unpacking the code. This command will need to be run again if you move the location of the Lisp-Stat directory.

```
wxls32.exe Autoload/win32com/server -RegServer
```

For now, the `-RegServer` flag must use the `-` flag, not `/`, and is case sensitive. This installs the server for the `ProgID` `XlispStat.application` described in Section 2.7.1. To unregister the server, run

```
wxls32.exe Autoload/win32com/server -UnregServer
```

This writeup still needs lots of work but is hopefully enough to get you started. Almost everything in this interface should be viewed as preliminary and subject to change.

2 Interface and Examples

2.1 Client Interface

The description of the client interface presented here is still very incomplete, but hopefully it is enough to get you started. The interface owes a lot to the Perl OLE interface; reading the short descriptions of this interface in [7] or [6] may be useful.

2.1.1 Initial Object Creation

Three functions are used for initial object creation.

`create-object` *cls-spec* `&key :server` [*Function*]

Creates a new object on the specified server; the default server is the local machine. The *cls-spec* can be a `ProgID` string or a class `GUID`. For example,

[1] See URL `http://www.stat.umn.edu/~luke/xls/projects/win32/win32com/win32com.zip`.
[2] See URL `http://www.stat.umn.edu/~luke/xls/projects/win32/win32.html`.

```
(create-object "Excel.application")
```

creates a new *Excel* server and returns its application object.

`create-object` takes a few additional keyword arguments but these are currently not particularly useful.

`get-object` *moniker* [*Function*]

Creates a new object as specified by the *moniker* string. Typically this string will be a file name. For example,

```
(get-object "...\\cars.xls")
```

starts an *Excel* server, opens the workbook in the `...\cars.xls` file, and returns the workbook object.

`get-active-object` *class-spec* [*Function*]

Returns the active object for *cls-spec* if one is registered. *class-spec* can be a `ProgID` or a class `GUID`.

2.1.2 Invoking Methods

Methods can be invoked for value or for side-effects only.

`invoke` *object name* `&rest` *args* [*Function*]

Invokes method *name* of *object* on the arguments and returns the resulting value. *name* can be a string or a symbol; method names are not case sensitive. For example,

```
(invoke engine :OpenDatabase "Northwind.mdb")
```

invokes the `OpenDatabase` method on a data base engine object. This takes a string naming the database file as its argument.

`invoke-no-value` *object name* `&rest` *args* [*Function*]

The same as `invoke`, but no value is returned.

Using `invoke-no-value` can be more efficient since it avoids the overhead of transferring the value from the server back to the client. It can also avoid errors if the server does not return a value—this seems to be the case for some methods in *Word*.

2.1.3 Reading and Setting Properties

`property` *object name* `&rest` *args* [*Function*]

Returns the value of the property *name* of *object*. *name* can be a string or a symbol; property names are not case sensitive. For example,

```
(property app :visible)
```

returns the current value of the `Visible` property of `app`. Additional arguments may be specified for index properties: The expression

```
(property wb :worksheets 1)
```

returns the first worksheet in the *Excel* workbook `wb`.

`(setf (property` *object name* {*arg*}*`)` *value*`)` [*Setf Method*]

Assigns *value* to property *name* in *object*. The expression

```
(setf (property add :visible) t)
```

sets the `Visible` property of `app` to `True`. This should cause the application object `app` to present a user interface if one is not already visible; servers usually are started without a visible user interface.

2.2 Client Examples

To use the COM client interface you need to know the methods and properties (and events for responding to events) a server object exposes. For MS applications you can find this out by reading books like [5]. For servers that provide type libraries you can use an object browser such as `OleView`, which is available from the **Downloads** section of the MS COM web page[3].

2.2.1 Accessing Excel

Suppose a file `cars.xls` contains some data on cars as an *Excel* workbook. The expression

6a ⟨*excel client example* 6a⟩≡ 6b ▷
```
(setf wb (get-object "...\\cars.xls"))
```

starts an *Excel* server and opens the workbook. These two steps can also be done separately by

6b ⟨*excel client example* 6a⟩+≡ ◁ 6a 6c ▷
```
(setf xl (create-object "Excel.application"))
(setf wb (invoke (property xl :workbooks) :open "...\\cars.xls"))
```

The expression

6c ⟨*excel client example* 6a⟩+≡ ◁ 6b 6d ▷
```
(setf sheet (property wb :worksheets 1))
```

obtains a reference to the first worksheet in the workbook and assigns it to the variable `sheet`.

Excel allows the rectangular region containing a specified cell and bordered by empty cells to be obtained by the `CurrentRegion` property of the cell. Thus, assuming that all the data are in a contiguous rectangular region, the data can be read from *Excel* into Lisp-Stat with the expression

6d ⟨*excel client example* 6a⟩+≡ ◁ 6c 6e ▷
```
(let ((range (property (property sheet :cells 1 1) :CurrentRegion)))
  (setf data  (property range :value)))
```

The result contained in the `data` variable is a Lisp-Stat matrix:

```
> data
#2A(("Mazda RX4"     21.0 6.0 160.0 110.0 ...)
    ("Mazda RX4 Wag" 21.0 6.0 160.0 110.0 ...)
    ...)
```

This matrix can then be used with any Lisp-Stat commands; for example a selection of three columns of the data can be placed in a rotatable plot with

6e ⟨*excel client example* 6a⟩+≡ ◁ 6d
```
(let ((cols (column-list data)))
  (setf car-spin (spin-plot (select cols '(1 3 4)))))
```

[3] See URL `http://www.microsoft.com/com`.

2.2.2 Using A Data Base

This example uses the *MS Data Access Object* (DAO) to access a data base. A typical interaction consists of starting an engine, opening a data base, obtaining a record set, and operating on the record set. The data base and record set should be closed once they are no longer needed:

7a ⟨*database client example* 7a⟩≡ 7e ▷
```
(let* ((engine (create-object "DAO.DBEngine.35"))
       (db (invoke engine :OpenDatabase ⟨database file name 7b⟩))
       (rs ⟨open database record set 7c⟩))
  (unwind-protect
      ⟨operate on the record set 7d⟩
    (invoke rs :close)
    (invoke db :close)))
```

This example uses the sample data base supplied with *MS Access*,

7b ⟨*database file name* 7b⟩≡ (7)
```
"\\Program Files\\Microsoft Office\\Office\\Samples\\Northwind.mdb"
```

The record set used corresponds to the `UnitPrice` variable in the `Products` table of the data base.

7c ⟨*open database record set* 7c⟩≡ (7a)
```
(invoke db :OpenRecordset "SELECT UnitPrice FROM Products")
```

The operation performed is to construct a histogram of the prices. This is done by first looping through the record set and extracting the prices into a list, and then making a histogram of the list.

7d ⟨*operate on the record set* 7d⟩≡ (7a)
```
(let ((prices nil))
  (loop
   (when (property rs :EOF) (return))
   (push (property (property rs :fields "UnitPrice") :value) prices)
   (invoke rs :MoveNext))
  (histogram prices))
```

An alternative way to open the database is to use `get-object` to start an *Access* server with the data base, and then obtain a reference to the database object as the `CurrentDB` property of the server object:

7e ⟨*database client example* 7a⟩+≡ ◁ 7a 7f ▷
```
(property (get-object ⟨database file name 7b⟩) :currentdb)
```

Record sets can be created from any SQL query supported by the engine. Some more examples:

7f ⟨*database client example* 7a⟩+≡ ◁ 7e
```
(invoke db :OpenRecordset
        "SELECT * FROM Employees ORDER BY LastName, FirstName"))
(invoke db :OpenRecordset
        (concatenate 'string
                     "SELECT CompanyName,Region,Country FROM Customers "
                     "WHERE Country = 'Canada' ORDER BY CompanyName"))
```

2.2.3 Using MS Agent

MS Agent is a silly little set of libraries that allow an application to pop up an animated character that speaks. Aside from being kind of fun, using this library from Lisp-Stat illustrates how to deal with some more difficult issues. This example is based on Haskell examples in [4]. The MS Agent software is available from the MS Agent web page[4].

[4]See URL `http://www.microsoft.com/msagent`.

```
[
  odl,
  uuid(A7B93C91-7B81-11D0-AC5F-00C04FD97575),
  helpstring("IAgent Interface"),
  dual,
  oleautomation
]
interface IAgent : IDispatch {
    [id(0x60020000)]
    HRESULT _stdcall Load(
                    [in] VARIANT vLoadKey,
                    [out] long* pdwCharID,
                    [out] long* pdwReqID);
    [id(0x60020001)]
    HRESULT _stdcall Unload([in] long dwCharID);
    [id(0x60020002)]
    HRESULT _stdcall Register(
                    [in] IUnknown* punkNotifySink,
                    [out] long* pdwSinkID);
    [id(0x60020003)]
    HRESULT _stdcall Unregister([in] long dwSinkID);
    [id(0x60020004)]
    HRESULT _stdcall GetCharacter(
                    [in] long dwCharID,
                    [out] IDispatch** ppunkCharacter);
    [id(0x60020005)]
    HRESULT _stdcall GetSuspended([out] long* pbSuspended);
};
```

Table 1: MS IDL for the `IAgent` Interface

Table 1 shows the MS IDL for the `IAgent` interface, which can be obtained from the `OleView` browser. Since the interface is marked as `dual` it is accessible from automation. One of the first methods we will need to use is the `Load` method. Unlike most methods used in automation, it has two `out` parameters and neither is marked as `retval`. This means that when called with Automation's invoke mechanism it will return no values but must be given two variant reference arguments through which the values for the `out` parameters are returned.

To start off, we need an agent server:

8a ⟨*MS Agent example* 8a⟩≡ 8b ▷

```
(setf agent-server (create-object "Agent.Server.2"))
```

Next, we need to load one of the standard characters. The function

8b ⟨*MS Agent example* 8a⟩+≡ ◁ 8a 9a ▷

```
(defun load-agent-character (server character-name)
  (let ((v1 (make-variant-ref nil VT_I4))
        (v2 (make-variant-ref nil VT_I4)))
    (invoke-no-value server :load character-name v1 v2)
    (let ((char-ref (variant-ref-value v1))
          (v3 (make-variant-ref nil VT_DISPATCH)))
```

```
      (invoke-no-value server :getcharacter char-ref v3)
      (values (variant-ref-value v3) char-ref))))
```

does this. `make-variant-ref` takes an initial value, here `nil`, and a type specification and returns a variant reference object. Two are needed for the call to the `Load` method of the server. The first receives the character reference index, which is extracted with `variant-ref-value`. The second `out` parameter receives the request ID and is ignored by this function. Many methods that initiate an asynchronous action return a request ID; I think this allows cancelation. Once the character index is available, a reference to the character itself is obtained by the `GetCharacter` method. Again the result is returned in a variant reference. The `load-agent-character` function returns both the character object reference and the index since the index is needed to unload the character.

The character `Merlin` is loaded by

9a ⟨*MS Agent example* 8a⟩+≡ ◁8b 9b▷

```
> (load-agent-character agent-server "Merlin.acs"))
#<IDISPATCH IAgentCharacterEx>
258
```

Other standard characters that may be installed are `Genie`, `Robby`, and `Peedy`.

Most of the operations on a character return a request ID through an `out` parameter. It is therefore more convenient to write some functions for them. This, and the wrapping of the `Load` method above, could and should be done automatically based on the IDL itself or on a Lisp-like IDL declaration, but for now you need to do it by hand. One approach to automating this is shown later in this section.

9b ⟨*MS Agent example* 8a⟩+≡ ◁9a 10a▷

```
(defun agent-character-show (char &optional fast)
  (let ((v (make-variant-ref nil VT_I4)))
    (invoke-no-value char :show (if fast t nil) v)
    (variant-ref-value v)))

(defun agent-character-speak (char text &optional (url ""))
  (let ((v (make-variant-ref nil VT_I4)))
    (invoke-no-value char :speak text url v)
    (variant-ref-value v)))

(defun agent-character-play (char action)
  (let ((v (make-variant-ref nil VT_I4)))
    (invoke-no-value char :play action url v)
    (variant-ref-value v)))

(defun agent-character-move-to (char x y speed)
  (let ((v (make-variant-ref nil VT_I4)))
    (invoke-no-value char :moveto x y speed v)
    (variant-ref-value v)))

(defun agent-character-think (char text)
  (let ((v (make-variant-ref nil VT_I4)))
    (invoke-no-value char :think text v)
    (variant-ref-value v)))

(defun agent-character-hide (&optional fast)
  (let ((v (make-variant-ref nil VT_I4)))
    (invoke-no-value char :hide (if fast t nil) v)
    (variant-ref-value v)))
```

The following expressions make use of these functions. They assume that a character is stored in the variable `c`.

10a ⟨*MS Agent example* 8a⟩+≡ ◁9b 11e▷

```
(agent-character-show c)
(agent-character-speak c "Hello")
(agent-character-play c "Greet")
(agent-character-speak c "Hello, World!")
(agent-character-play c "Wave")
(agent-character-move-to c 700 100 50)
(agent-character-think c "Hello")
(agent-character-think c "Hello")
(invoke agent-server :unload 258)
```

Another approach to handling an interface like this would be to wrap references in Lisp-Stat objects. A prototype for the server wrapper could be defined to contain a slot for the COM reference.

10b ⟨`agent-server` *prototype* 10b⟩≡ 10c▷

```
(export 'agent-server)
(defproto agent-server '(com-reference))
```

The initialization method creates a new COM server and installs its reference.

10c ⟨`agent-server` *prototype* 10b⟩+≡ ◁10b 10d▷

```
(defmeth agent-server :isnew ()
  (setf (slot-value 'com-reference) (create-object "Agent.Server.2")))
```

To simplify connecting Lisp-Stat methods to the COM methods we can define a macro `define-agent-method`. Using this macro, which is given below, the `Load` method and the `GetCharacter` method from the IDL are connected to methods for `agent-server` by

10d ⟨`agent-server` *prototype* 10b⟩+≡ ◁10c 10e▷

```
(define-agent-method agent-server base-load
  (file (:out char-ref VT_I4) (:out req-id VT_I4))
  :load)
(define-agent-method agent-server get-character
  (char-ref (:out char VT_DISPATCH))
  :GetCharacter)
```

The names for these methods are internal symbols in the `MS-AGENT` package rather than keyword symbols since they are intended to be used only within the package.

A method that combines these two and also wraps the COM character reference in a Lisp-Stat object is defined as

10e ⟨`agent-server` *prototype* 10b⟩+≡ ◁10d

```
(defmeth agent-server :load-character (file)
  (let* ((char-ref (send self 'base-load file))
         (char (send self 'get-character char-ref)))
    (send agent-character :new char char-ref self)))
```

This is the public character loading method.

Loading a character that is already loaded does not seem to work; it might be a good idea to have the server keep track of which characters are loaded and only load new ones if they are not loaded already. This sort of thing would probably require a reference/lock count on the character. The server might also provide a higher level character naming interface so users would not need to remember the `.acs` file name extension.

The character representation is defined by a prototype containing slots for a COM reference, the server that produced the character, and the character reference index returned by the server `Load` method. This prototype is not exported since new characters should only be created by the sending a server a `:load-character` message.

11a ⟨`agent-character` *prototype* 11a⟩≡ (12e) 11b ▷
```
(defproto agent-character '(com-reference server char-ref))
```

The initialization method is

11b ⟨`agent-character` *prototype* 11a⟩+≡ (12e) ◁ 11a 11c ▷
```
(defmeth agent-character :isnew (comref char-ref server)
  (setf (slot-value 'com-reference) comref)
  (setf (slot-value 'char-ref) char-ref)
  (setf (slot-value 'server) server))
```

Since the character contains a reference to its server as well as its reference index, unloading can now be handled by a character method:

11c ⟨`agent-character` *prototype* 11a⟩+≡ (12e) ◁ 11b 11d ▷
```
(defmeth agent-character :unload ()
  (let ((server (send (slot-value 'server) :slot-value 'com-reference)))
    (invoke-no-value server :unload (slot-value 'char-ref))))
```

Here are some basic character methods:

11d ⟨`agent-character` *prototype* 11a⟩+≡ (12e) ◁ 11c
```
(define-agent-method agent-character :show
  ((:optional fast nil) (:out req-id VT_I4)))
(define-agent-method agent-character :hide
  ((:optional fast nil) (:out req-id VT_I4)))
(define-agent-method agent-character :position
  ((:out x VT_I4) (:out y VT_I4)) :GetPosition)
(define-agent-method agent-character :set-position (x y) :SetPosition)
(define-agent-method agent-character :move-to
  (x y speed (:out req-id VT_I4)) :MoveTo)
(define-agent-method agent-character :think (text (:out req-id VT_I4)))
(define-agent-method agent-character :play (action (:out req-id VT_I4)))
(define-agent-method agent-character :speak
  (text (:optional url "") (:out req-id VT_I4)))
```

Here are a few examples:

11e ⟨*MS Agent example* 8a⟩+≡ ◁ 10a
```
(setf s (send ms-agent:agent-server :new))
(setf c (send s :load-character "Merlin.acs"))
(send c :show)
(send c :speak "Hello")
(send c :play "Greet")
(send c :speak "Hello, World!")
(send c :play "Wave")
(send c :position)
(send c :set-position 100 100)
(send c :move-to 700 100 50)
(send c :think "Hello")
(send c :unload)
```

Finally we need to define the `define-agent-method` macro. This definition illustrates how the information available in IDL can be used to construct a Lisp interface. The macro is called with the object, the Lisp-Stat method name, an argument list, and an optional COM method name as arguments; the COM method name defaults to the Lisp-Stat one.

11f ⟨define-agent-method *macro* 11f⟩≡ (12e)

```
(defmacro define-agent-method (object name args &optional (comname name))
  (let ((invars nil)
        (optargs nil)
        (outargs nil)
        (callargs nil))
    ⟨process args into invars, optargs, outargs, and callargs 12a⟩
    (let ((arglist ⟨construct the agent method lambda list 12c⟩)
          (binds ⟨construct bindings for the out parameters 12b⟩)
          (vals ⟨construct the agent method value expressions 12d⟩))
      `(defmeth ,object ,name ,arglist
         (let ,binds
           (invoke-no-value (slot-value 'com-reference) ',comname ,@callargs)
           (values ,@vals))))))
```

The argument list entries can be symbols or lists. Symbols are treated as standard `in` arguments. Lists must start with `:out` or `:optional`. `:out` lists represent `out` parameters; the `:out` symbol must be followed by a variable symbol and a `VARIANT` type. `:optional` lists represent optional parameters; the second term in the list must be a symbol naming the variable and the third must be an expression for the default value. `:out` arguments can appear anywhere; all arguments following an `:optional` argument must be either `:out` or `:optional`. None of this is checked. The argument list is processed into the standard `in` arguments and the `out` and optional arguments; the arguments for the COM method invocation are also accumulated.

12a ⟨*process* `args` *into* `invars`, `optargs`, `outargs`, *and* `callargs` 12a⟩≡ (11f)

```
(dolist (a args)
  (if (symbolp a)
      (progn (push a invars) (push a callargs))
    (case (first a)
          (:optional (push (rest a) optargs)
                     (push (second a) callargs))
          (:out (push (rest a) outargs)
                (push (second a) callargs)))))
(setf invars (nreverse invars))
(setf optargs (nreverse optargs))
(setf outargs (nreverse outargs))
(setf callargs (nreverse callargs))
```

The invocation is contained in a `let` binding that creates references for returning the `out` parameters. The bindings are constructed by

12b ⟨*construct bindings for the* `out` *parameters* 12b⟩≡ (11f)

```
(mapcar (lambda (x)
          `(,(first x) (make-variant-ref nil ,(second x))))
        outargs)
```

The lambda list for the method definition is built by

12c ⟨*construct the agent method lambda list* 12c⟩≡ (11f)

```
(if optargs `(,@invars &optional ,@optargs) invars)
```

and the expressions for the values to be returned by the method are constructed by

12d ⟨*construct the agent method value expressions* 12d⟩≡ (11f)

```
(mapcar (lambda (x) `(variant-ref-value ,(first x))) outargs)
```

The agent interface is contained in the file `msagent.lsp`[5].

[5] See URL `msagent.lsp`.

12e ⟨*msagent.lsp* 12e⟩≡
```
(defpackage "MICROSOFT-AGENT" (:nicknames "MS-AGENT") (:use "XLISP" "COM"))
(in-package "MS-AGENT")
⟨define-agent-method macro 11f⟩
⟨agent-server prtotype (never defined)⟩
⟨agent-character prototype 11a⟩
```

2.3 Reflection

The COM interface provides some reflection capabilities for servers that provide type libraries. The interface does provide access to most of the type library and type info methods, but this is too low level to be worth documenting here. I still need to come up with a useful higher level interface. For the moment, this section presents a couple of useful routines. They are not part of the interface—you need to cut and past them into the listener or something of that sort before they can be used.

Here is a function that lists all the methods (actually methods and properties) the interface of an Automation object exposes.

13a ⟨*reflection* 13a⟩≡ 13b ▷
```
(defun com-methods (ref &optional full)
  (let ((info (idispatch-type-info ref)))
    (unless info (error "no type information available"))
    (com-info-methods info full)))
```

It uses the function `com-info-methods` to get the information it needs from a type information object.

13b ⟨*reflection* 13a⟩+≡ ◁ 13a 13c ▷
```
(defun com-info-methods (info full)
  (let* ((desc (itypeinfo-function-descriptions info))
         (names (mapcar #'ninth desc)))
    (if full names (mapcar #'first names))))
```

The function `com-info-methods` returns either a list of method names or a list of lists with each list containing the method name followed by its arguments.

As an example, suppose we open an *Excel* workbook and ask it for information on its methods.

```
> (setf wb (get-object "...\\cars.xls"))
#<IDISPATCH _Workbook>
> (com-methods wb)
("QueryInterface" "AddRef" "Release" "GetTypeInfoCount" "GetTypeInfo"
 "GetIDsOfNames" "Invoke" "Application" "Creator" "Parent"
 "AcceptLabelsInFormulas" "AcceptLabelsInFormulas" "Activate" ...)
```

The first seven methods are not actually Automation-callable methods; they are part of the Automation interface. (I'm not sure if they are always in the first seven positions; if so, `com-info-methods` could just drop them).

Instead of using the `full` argument, we could extract the signature. One (inefficient) way to do this uses

13c ⟨*reflection* 13a⟩+≡ ◁ 13b 14 ▷
```
(defun com-method-signature (ref name)
  (let ((info (idispatch-type-info ref)))
    (unless info (error "no type information available"))
    (com-info-method-signature info name)))

(defun com-info-method-signature (info name)
```

```
    (ninth (find (string name) (itypeinfo-function-descriptions info)
                 :test #'string-equal
                 :key #'(lambda (x) (first (ninth x))))))
```

For the `Activate` method of our workbook this gives

```
  > (com-method-signature wb "Application")
  ("Application")
```

So the method takes no arguments.

A similar set of functions can be used to extract the method names (i.e. the events) or signatures for an event source:

14 ⟨*reflection* 13a⟩+≡ ◁ 13c

```
  (defun com-events (object &optional full)
    (let* ((lib (itypeinfo-type-lib (idispatch-type-info object)))
           (iid (first (get-source-interfaces object))))
      (when iid
        (let ((info (itypelib-type-info-of-guid lib iid)))
          (when (eq :dispatch (itypeinfo-kind info))
            (com-info-methods info full))))))

  (defun com-event-signature (object name)
    (let* ((lib (itypeinfo-type-lib (idispatch-type-info object)))
           (iid (first (get-source-interfaces object))))
      (when iid
        (let ((info (itypelib-type-info-of-guid lib iid)))
          (when (eq :dispatch (itypeinfo-kind info))
            (com-info-method-signature info name))))))
```

For the workbook:

```
  > (com-events wb)
  ("QueryInterface" "AddRef" "Release" "GetTypeInfoCount" "GetTypeInfo"
   "GetIDsOfNames" "Invoke" "Open" "Activate" "Deactivate" "BeforeClose"
   "BeforeSave" "BeforePrint" "NewSheet" "AddinInstall" "AddinUninstall"
   "WindowResize" "WindowActivate" "WindowDeactivate" "SheetSelectionChange"
   "SheetBeforeDoubleClick" "SheetBeforeRightClick" "SheetActivate"
   "SheetDeactivate" "SheetCalculate" "SheetChange")
  > (com-event-signature wb :newsheet)
  ("NewSheet" "Sh")
```

2.4 Value Conversion

The COM Automation subset passes arguments and returns values as data of type `VARIANT`. These are self-describing structures containing a type specifier and a value, much like Lisp values. The set of possible types places some restrictions on data that can be transferred (at least currently; COM+ will alleviate some of these restrictions [2]) but also allows many types of values to be converted automatically from Lisp to `VARIANT` and back.

2.4.1 Automatic Conversion for Basic Value Types

Table 2 shows the correspondence used for automatic conversion of `VARIANT` values, e.g. method results, to Lisp. Most `VARIANT` types fit reasonably into Lisp types. One slightly problematic point is the handling of the `VT_EMPTY` and `VT_NULL` types. This is discussed further in Section 2.4.2

Default conversion from Lisp values to `VARIANT` values is shown in Table 3.

Description	Variant Type	Lisp Type
Array	VT_ARRAY	(array t)
Empty	VT_EMPTY	(eql :empty)
Null	VT_NULL	(eql :null)
1-Byte unsigned int	VT_UI1	fixnum
2-Byte signed int	VT_I2	fixnum
4-Byte signed int	VT_I4	fixnum
4-Byte real	VT_R4	float
8-Byte real	VT_R8	float
Currency	VT_CY	rational
String	VT_BSTR	string
Error Code	VT_ERROR	fixnum
Boolean	VT_BOOL	(or null (eql t))
Date	VT_DATE	float
Automation Object	VT_DISPATCH	idispatch
Variant	VT_VARIANT	variant
COM Object	VT_UNKNOWN	iunknown

Table 2: VARIANT to Lisp Conversion

Description	Lisp Type	Variant Type
Array	(and array (not string))	VT_ARRAY of VARIANT
Symbol nil	null	VT_BOOL False
Symbol t	(eql t)	VT_NULL True
Fixnum	fixnum	VT_I4
Float	float	VT_R8
String	string	VT_BSTR
Automation Object Reference	idispatch	VT_DISPATCH
Variant	variant	VT_VARIANT
COM Object Reference	iunknown	VT_UNKNOWN
COM Server Object	inherits from com-server	VT_DISPATCH

Table 3: Lisp to VARIANT Conversion

2.4.2 Empty and Null Values

It might seem natural to return values of type `VT_EMPTY` and `VT_NULL` as `nil`, but this would conflict with representing the boolean `False` value as `nil`. I'm not sure what the best approach is, but here is what I do for now. The conversions for `VT_NULL` and `VT_EMPTY` actually use the values of the special (i.e. dynamically scoped) variables `*null-variant-value*` and `*empty-variant-value*`. The default values are the keyword symbols `:null` and `:empty`. To change this you could re-bind these variables. For example, to return the keyword symbol `:missing` for empty cells in a spread sheet you could use something like

```
(let ((*empty-variant-value* :missing))
  (property sheet :cells i j))
```

2.4.3 Explicit Variant Passing and Conversion

Occasionally it is useful to force an argument value to be passed as a particular `VARIANT` type. This can be done using the functions described here.

`make-variant` *arg* `&optional` *type* [*Function*]

Constructs a `VARIANT` with value *arg* and type *type*. If *type* is omitted, then the default from Table 3 is used.

`variant-value` *var* [*Function*]

Returns the Lisp value of `VARIANT` *var* as given in 2.

`variant-change-type` *var type* [*Function*]

Creates a new `VARIANT` with type *type* and value corresponding to the value of the `VARIANT` *var*

`variant-value-as` *var type* [*Function*]

This is shorthand for `(variant-value (variant-change-type` *var type*`))`.

As one example, `VT_DATE` values are stored as a floating point number containing days since December 30, 1899 (i.e. for that date the value is zero):

```
> (variant-value-as (make-variant "12/30/1899") VT_DATE)
0.0
> (variant-value-as (make-variant "January 17, 2000") VT_DATE)
36542.0
```

Another case where explicit variant creation may be useful is for currency (`VT_CY`) values. A currency value is stored as a 64-bit integer n that represents $n/10,000$ currency units. On conversion to Lisp `VT_CY` values are converted to rationals using `bignum`'s if necessary, so any `VT_CY` value is representable in Lisp. Small currency values can be passed as `fixnum` arguments or `float` arguments; the server will convert them to currency values. But large currency values requiring `bignums` will not be handled by automatic conversion; here again explicit variants can be used.

```
> (variant-value (make-variant 100000000000 VT_CY))
100000000000
```

2.4.4 COM References and Servers

COM objects returned by COM methods can be passed as arguments to methods that accept them. COM servers created in Lisp-Stat can also be passed as arguments. COM servers are discussed further below.

2.5 Responding to Events

Suppose we start an *Excel* application, load a worksheet, and make the application visible and available for user interaction. The first worksheet in `wb` is stored in `sheet` and a `spin-plot` of the data is made and stored in `car-spin`

17a ⟨*events example* 17a⟩≡ 17b ▷

```
(setf xl (create-object "Excel.application"))
(setf wb (invoke (property xl :workbooks) :open "...\\cars.xls"))
(setf (property xl :visible) t)
(setf sheet (property wb :worksheets 1))
(let* ((range (property (property sheet :cells 1 1) :CurrentRegion))
       (data (property range :value))
       (cols (column-list data)))
  (setf car-spin (spin-plot (select cols '(1 3 4)))))
```

At this point we might like to edit the data in the spread sheet, and it would be nice if the plot of the data could be updated automatically to reflect any changes we make. This can be arranged by registering an event listener with the `sheet` object. An event listener for `sheet` is constructed and connected with

17b ⟨*events example* 17a⟩+≡ ◁ 17a 17c ▷

```
(setf sheet-listener (send event-server :new sheet))
(send sheet-listener :connect)
```

Once connected, this object receives notifications from `sheet` whenever certain events occur as a result of user interaction or executing a program. These events include changes to data and changes to the focus cell, among others. A listing of event names for *Excel* can be found in a reference such as [5], by using a COM browser, or using the utilities described in Section 2.3. The event listener is now a server for which the worksheet is the client.

Notifications are ignored unless the event listener defines an appropriate method. The method name is specified by the source interface for the `Worksheet` object, and the name of the method called when the sheet's data changes is `Change`. Here is a simple definition of a `Change` method for updating the plot when the sheet's data changes:

17c ⟨*events example* 17a⟩+≡ ◁ 17b

```
(defmeth sheet-listener :change (range)
  (let ((cell (property sheet :cells 1 1)))
    (let ((data (property (property cell :CurrentRegion) :value)))
      (let ((cols (column-list data)))
        (send car-spin :clear :draw nil)
        (send car-spin :add-points (select cols '(1 3 4)) :draw nil)
        (send car-spin :adjust-to-data)))))
```

This definition ignores the `range` argument that reflects the cells that have changed and simply copies all the data from the spread sheet. A more sophisticated approach would make use of the `range` argument to reduce the amount of data transferred.

The event handling mechanism is built on the server interface which is described next.

2.6 Server Interface

The server interface is even more preliminary than everything else described here, so take it with a grain of salt.

The highest level server interface is implemented by two Lisp-Stat prototype objects.

2.6.1 Automation Server Prototype

`auto-server` [*Prototype*]

Prototype object for COM Automation servers.

`send` *server* `:isnew &optional` *name* [*Method*]

The initialization argument takes a server name string *name* as an optional argument; the default server name is `"Xlisp-Stat"`.

`send` *server* `:server-name` [*Method*]

Returns the server name of *server*.

`send` *server* `:add-auto-property` *com-name get-name* `&optional` *set-name* [*Method*]

Registers a COM property *com-name*. A property `get` will send the *get-name* message; a `put` will send the *set-name* message. If *set-name* is omitted then the property is registered as read-only.

`send` *server* `:add-auto-method` *com-name name* `&optional` *for-value* [*Method*]

Registers a COM method *com-name*. The COM method is handled by sending the object the *name* message. If *for-value* is true, the default, a value is returned to the calling client; if it is false, no value is returned.

`send` *server* `:register-active-object` *class-spec* [*Method*]

Registers *server* as the active object for *class-spec*. *class-spec* can be a class `GUID` or a `ProgID` string. A server can be registered as the active object for several *class-spec* values.

`send` *server* `:revoke-active-object &optional` *class-spec* [*Method*]

If *class-spec* is provided, the registration of *server* as the active object for *class-spec* is revoked. Otherwise all active object registrations of *server* are revoked.

The `auto-server` prototype has one pre-registered read-only property, `Name`, which returns the server name.

The `auto-server` prototype has a number of internal methods that should not be redefined: `:MAKE-SERVER`, `:INVOKE`, `:GET-IDS-OF-NAMES`, `:SERVER`, and `:AUTO-METHODS`. Eventually these will either be hidden using the package mechanism or documented as a lower level mechanism.

2.6.2 Class Factories

When COM creates a new object in response to a call to `create-object` it does so by finding a registered class factory for the specified class and asking the class factory to create a new object. The `class-factory` prototype provides the required functionality.

`class-factory` [*Prototype*]

Class factory prototype.

`(send` *factory* `:isnew` *proto* `&key :singleton)` [*Method*]

The initialization method requires an object *proto* as its argument. This object should inherit from `auto-server`. When `:singleton` is omitted or `nil`, *proto* is sent the `:new` message with no arguments to produce a new server; a reference to this server is returned to the client requesting the object creation. If `:singleton` is true, then *proto* itself is returned.

`(send` *factory* `:register` *cls-spec*`)` [*Method*]

Registers *factory* as the class factory for *cls-spec*, a class `GUID` or `ProgID`. A factory can be registered as the factory for several *cls-spec*s but I don't think there is a way to tell which one is requesting object creation.

`(send` *factory* `:revoke &optional` *cls-spec*`)` [*Method*]

Revoke the class factory registration of *factory* for *cls-spec* or for all classes if *cls-spec* is omitted or `nil`.

2.6.3 Registering a COM Server

To be able to use a Lisp-Stat class factory from another application, such as *Excel*'s VBA, some information has to be entered into the registry. This information includes the mapping from a `ProgID` string to a `CLSID` and the command used to start the server process (or information about the server DLL, but Lisp-Stat is not yet able to produce a DLL-based server). The functions described here do the required registry manipulation. This interface is based on the C++ functions `RegisterServer` and `UnregisterServer` provided in [2].

`register-com-server` *command clsid friendly-name progid vi-progid* [*Function*]

Performs the registry operations needed to register the server. *command* is a string with the command to start the server, *clisd* is a `CLSID` string, and *friendly-name* is a string used for labeling the registered class in class browsers. The *progid* string is a `ProgID` that may include a version number, and *vi-progid* is the version-independent `ProgID` string. For the initial version of the Lisp-Stat server *progid* would be `"XlispStat.application.1"` and *vi-progid* would be `"XlispStat.application"`.

`unregister-com-server` *clsid progid vi-progid* [*Function*]

Removes the server registration for the specified *clsid*, *progid*, and *vi-progid* from the registry.

If you create your own server you will need a `CLSID`, a `GUID` (Globally Unique IDentifier) used for a COM class. You can create one with `create-guid`:

```
> (create-guid)
#<GUID {776975F0-CE9B-11D3-9F0F-000502DB4890}>
```

The string representation required for registration is the string between the braces, `"{77...}"`. Do *not* use the one shown here—you need to generate your own.

2.7 Server Examples

2.7.1 A Lisp-Stat Server

This example presents the server for `XlispStat.application` that is registered when the server in the distribution is installed. Other servers can be built by adapting this example. The implementation is in the file `server.lsp`. Since this is part of the COM interface, it is placed in the `WIN32-COM` package; you should put your own code in a different package.

19 ⟨*server.lsp* 19⟩≡
```
(require "win32")
(require "win32com")
(in-package "WIN32-COM")
⟨Lisp-Stat server registration 20a⟩
⟨Lisp-Stat server implementation 20c⟩
⟨Lisp-Stat class factory implementation 20d⟩
⟨Lisp-Stat class factory registration 21d⟩
```

The registration code attempts to be moderately close to the recommended mechanism for registering server applications. When the server file is loaded, the command line arguments used to start the process are checked. If `-RegServer` is one of those arguments, the server is registered and the process exits.

20a ⟨*Lisp-Stat server registration* 20a⟩≡ (19) 20b ▷
```
(when (find "-RegServer" *command-line* :test #'string=)
  (let ((command (format nil "~awxls32.exe Autoload\\win32com\\server"
                         *default-path*))
        (clsid "{FB4C2CC0-60EF-11D3-8E98-444553540000}")
        (friendly-name "XlispStat COM Server")
        (progid "XlispStat.application.1")
        (vi-progid "XlispStat.application"))
    (register-com-server command clsid friendly-name progid vi-progid))
  (exit))
```

The `CLISD` used here is specific to the Lisp-Stat server and should not be re-used. To generate your own `CLSID` you can call the function `create-guid`.

Unregistration is analogous: if `-UnregServer` is in the command line arguments, the server is unregistered and the process exits.

20b ⟨*Lisp-Stat server registration* 20a⟩+≡ (19) ◁ 20a
```
(when (find "-UnregServer" *command-line* :test #'string=)
  (let ((clsid "{FB4C2CC0-60EF-11D3-8E98-444553540000}")
        (progid "XlispStat.application.1")
        (vi-progid "XlispStat.application"))
    (unregister-com-server clsid progid vi-progid))
  (exit))
```

The Lisp-Stat application server is a singleton server; its class factory always returns a reference to the same server object.

20c ⟨*Lisp-Stat server implementation* 20c⟩≡ (19) 20e ▷
```
(defparameter *xlispstat-auto-server* (send auto-server :new))
```

20d ⟨*Lisp-Stat class factory implementation* 20d⟩≡ (19)
```
(defparameter *xlispstat-class-factory*
  (send class-factory :new *xlispstat-auto-server* :singleton t))
```

The server provides a number of COM methods. The first is a `Load` method for loading a specified file.

20e ⟨*Lisp-Stat server implementation* 20c⟩+≡ (19) ◁ 20c 20f ▷
```
(defmeth *xlispstat-auto-server* :load (file)
  (load file))
(send *xlispstat-auto-server* :add-auto-method :load :load)
```

Next, there are three methods for evaluating an expression passed in as a string. The Automation names for these methods are `Eval`, `EvalNoValue`, and `EvalAllValues`. The methods return a single value, no value, or a vector of all values, respectively. ****AllValues–check no values (zero-length array) in VBA–works when WXLS32 is the client.

20f ⟨*Lisp-Stat server implementation* 20c⟩+≡ (19) ◁20e 21a▷

```
(defmeth *xlispstat-auto-server* :eval (expr)
  (eval (read-from-string expr)))
(send *xlispstat-auto-server* :add-auto-method :eval :eval)
(send *xlispstat-auto-server* :add-auto-method :evalnovalue :eval nil)

(defmeth *xlispstat-auto-server* :eval-all-values (expr)
  (coerce (multiple-value-list (eval (read-from-string expr))) 'vector))
(send *xlispstat-auto-server* :add-auto-method :evalallvalues :eval-all-values)
```

Three analogous methods are provided for calling a function. The function is specified as a string; the arguments are passed using the standard conversions. The function string can name a function; it can also be a lambda expression.

21a ⟨*Lisp-Stat server implementation* 20c⟩+≡ (19) ◁20f 21b▷

```
(defmeth *xlispstat-auto-server* :call (fun &rest args)
  (apply (read-from-string fun) args))
(send *xlispstat-auto-server* :add-auto-method :call :call)
(send *xlispstat-auto-server* :add-auto-method :callnovalue :call nil)

(defmeth *xlispstat-auto-server* :call-all-values (fun &rest args)
  (coerce (multiple-value-list (apply (read-from-string fun) args)) 'vector))
(send *xlispstat-auto-server* :add-auto-method :callallvalues :call-all-values)
```

The `Create` method takes the name of a server prototype, sends it the `:new` message with the specified arguments, and returns a reference to the new server.

21b ⟨*Lisp-Stat server implementation* 20c⟩+≡ (19) ◁21a 21c▷

```
(defmeth *xlispstat-auto-server* :create (name &rest args)
  (apply #'send (symbol-value (read-from-string name)) :new args))
(send *xlispstat-auto-server* :add-auto-method :create :create)
```

The `Visible` property can be used to query or set whether the Lisp-Stat application window is visible or not. Initially it is created invisible by COM, and the server will exit once all references to its objects have been released as long as the main window is not visible.

21c ⟨*Lisp-Stat server implementation* 20c⟩+≡ (19) ◁21b

```
(defmeth *xlispstat-auto-server* :visible (&optional (vis nil set))
  (if set (msw-main-frame-visible vis) (msw-main-frame-visible)))
(send *xlispstat-auto-server* :add-auto-property :visible :visible :visible)
```

The final step in the file is to register the class factory.

21d ⟨*Lisp-Stat class factory registration* 21d⟩≡ (19)

```
(send *xlispstat-class-factory* :register "XlispStat.application")
```

2.7.2 Using the Server from Excel

The roles of *Excel* and Lisp-Stat in the example of Section 2.2.1 can be reversed by using the Lisp-Stat application server object from an *Excel* client. In VBA we can define a subroutine `SpinPlot` to take three arguments specifying three columns of the first worksheet in the first workbook and produce a Lisp-Stat spinplot.

21e ⟨*VBA* `SpinPlot` *subroutine* 21e⟩≡

```
Sub SpinPlot(x, y, z)
    Dim xls As Object, sheet As Worksheet, data As Range
    ⟨VBA SpinPlot body 22a⟩
End Sub
```

The first step is to create a new Lisp-Stat application object and save its reference in the variable `xls`.

22a ⟨*VBA* `SpinPlot` *body* 22a⟩≡ (21e) 22b▷
```
Set xls = CreateObject("XlispStat.application")
```

Then we select the data to be used.

22b ⟨*VBA* `SpinPlot` *body* 22a⟩+≡ (21e) ◁22a 22c▷
```
Set sheet = Application.Workbooks(1).Worksheets(1)
Set data = sheet.Cells(1, 1).CurrentRegion
```

By convention, COM servers typically are started with no visible user interface. To make the server application visible, its `Visible` property must be set to `True`.

22c ⟨*VBA* `SpinPlot` *body* 22a⟩+≡ (21e) ◁22b 22d▷
```
xls.Visible = True
```

The basic objective is to call the Lisp-Stat `spin-plot` function with the data from the specified columns. This function expects a list of three lists or vectors, but data column values produced by *Excel* are single column matrices. The following expression uses the Lisp-Stat application object's `EvalNoValue` method to define a function that receives the data in a form natural to Excel, converts the data to the form needed by `spin-plot`, and makes the call. The `EvalNoValue` call passes a string argument to Lisp-Stat where the contents of the string are read and evaluated. The value is ignored; this avoids receiving an error since the value returned by `defun` is a symbol, and there is no default conversion for symbols (other than `nil` and `t`).

22d ⟨*VBA* `SpinPlot` *body* 22a⟩+≡ (21e) ◁22c 22e▷
```
xls.EvalNoValue "(defun sp (x y z)" & _
                "  (flet ((as-seq (x)" & _
                "           (compound-data-seq x)))" & _
                "    (spin-plot (list (as-seq x)" & _
                "                     (as-seq y)" & _
                "                     (as-seq z)))))"
```

The final step is to call the new function with the contents of the specified columns. This uses the `CallNoValue` method of the application server.

22e ⟨*VBA* `SpinPlot` *body* 22a⟩+≡ (21e) ◁22d
```
xls.CallNoValue "sp", data.Columns(x).Value, _
                      data.Columns(y).Value, _
                      data.Columns(z).Value
```

An alternate approach would be to create a string containing the lambda expression for the function `sp` and passing that string to the `CallNoValue` method. This would avoid cluttering the name space with the function `sp`.

2.7.3 Distributed Computing

This example illustrates the use COM in a distributed environment. A simulation is to be split over m machines. The particular simulation task used as an illustration is to evaluate the expected value of the sample median for a sample of n χ^2_d random variables using a total simulation sample size of mk with k simulations run on each machine. The simulation for each machine is handled by the simple function

22f ⟨*distributed simulator* 22f⟩≡ (24c) 23a▷
```
(defun sim (n k d)
  (let ((val (make-array k)))
    (dotimes (i k (mean val))
      (setf (aref val i) (median (chisq-rand n d))))))
```

The distributed program uses a supervisor/worker model [3] in which a supervisor program starts up m worker programs on each of m machines and then collects their results. The workers are implemented by COM servers based on the prototype defined by

23a ⟨*distributed simulator* 22f⟩+≡ (24c) ◁22f 23b▷

```
(defproto median-simulator () () auto-server)
```

The simulation on a particular machine is carried out by the `:run` method of the simulation server.

23b ⟨*distributed simulator* 22f⟩+≡ (24c) ◁23a 23d▷

```
(defmeth median-simulator :run (n k d receiver)
  (flet ((runner (n k d rcv) (setf (property rcv :value) (sim n k d))))
    (async-call #'runner n k d receiver)))
```

COM does not yet provide support for asynchronous calls (this will change with COM+ in Windows 2000), so the `:run` method must start the computation and then return immediately. Ideally this would be done by creating a separate thread to carry out the computation, but this is currently not possible since Lisp-Stat does not yet support multiple threads. Instead an asynchronous call mechanism provided by the `async-call` function can be used. This mechanism places the call on the event queue, where it is processed in idle time.

23c ⟨*asynchronous function call* 23c⟩≡ (24c)

```
(defun async-call (fun &rest args)
  (push (cons fun args) *event-queue*)
  (values))
```

To make the Lisp-Stat `:run` method available as a COM method, it needs to be registered by

23d ⟨*distributed simulator* 22f⟩+≡ (24c) ◁23b

```
(send median-simulator :add-auto-method :run :run)
```

The last argument to the `:run` method is a value receiver, a COM object owned by the supervisor that accepts the value of the computation as its `Value` property. This receiver object is also a COM server. It contains a semaphore object that is used to signal when the receiver has been given a value.

23e ⟨*distributed value receiver* 23e⟩≡ (24c) 23f▷

```
(defproto receiver '(semaphore value) () auto-server)
```

The initialization method creates the semaphore locked (count of zero).

23f ⟨*distributed value receiver* 23e⟩+≡ (24c) ◁23e 23g▷

```
(defmeth receiver :isnew ()
  (call-next-method)
  (setf (slot-value 'semaphore) (make-semaphore 0)))
```

The receiver's `:value` method waits until the semaphore is released before retrieving the contents of the `value` slot.

23g ⟨*distributed value receiver* 23e⟩+≡ (24c) ◁23f 23h▷

```
(defmeth receiver :value ()
  (wait-semaphore (slot-value 'semaphore))
  (slot-value 'value))
```

The `:set-value` method places a new value in the `value` slot and then releases the semaphore.

23h ⟨*distributed value receiver* 23e⟩+≡ (24c) ◁23g 23i▷

```
(defmeth receiver :set-value (v)
  (setf (slot-value 'value) v)
  (release-semaphore (slot-value 'semaphore)))
```

Together these two methods make up the `Value` property of the corresponding COM object.

23i ⟨*distributed value receiver* 23e⟩+≡ (24c) ◁23h

```
(send receiver :add-auto-property :value :value :set-value)
```

This receiver is intended to be used only once; otherwise an additional lock would be needed to insure that it is not written before the value has been read.

It is important that the blocking wait for a value occur locally, not in a COM call, since COM is designed for synchronous calls and a wait might be interpreted as a communication failure.

The supervisor is implemented by two functions. The first function starts the simulations. It takes the simulation parameters and a list of machine names as arguments. For each machine the local function `start` creates a receiver object and starts a Lisp-Stat server on the specified machine. The server is asked to load the file `medsim` containing the code for the simulation, and is then asked to create a new simulator. Finally, the `:run` method on this simulator is called to start its work and the receiver is returned. The function `start` is applied to each machine on the machine list and a list of the receivers is returned.

24a ⟨*distributed supervisor* 24a⟩≡ (24c) 24b ▷

```
(defun start-simulations (n k d machines)
  (flet ((start (mach)
           (let ((rcv (send receiver :new))
                 (app (create-object "XlispStat.application" :server mach)))
             (invoke app :load "medsim")
             (let ((sim (invoke app :create "median-simulator")))
               (invoke sim :run n k d rcv))
             rcv)))
    (map 'list #'start machines)))
```

Once `start-simulations` has been called, the simulations are running in parallel on their respective machines and the supervisor needs to collect the results. This is done by the function `collect-values`. This function requests the value for each receiver and returns a list of the results.

24b ⟨*distributed supervisor* 24a⟩+≡ (24c) ◁ 24a

```
(defun collect-values (receivers)
  (flet ((collect (rcv) (send rcv :value)))
    (map 'list #'collect receivers)))
```

To carry out a simulation using $m = 2$ machines, $k = 10,000$ simulations per machine, samples of size $n = 10$, and $d = 5$ degrees of freedom for the populations sampled, the supervisor would use an expression of the form

```
(let ((machines (list "192.168.1.3" "192.168.1.10")))
  (collect-values (start-simulations 10 10000 5 machines)))
```

The code for this example is in the file `medsim.lsp`[6].

24c ⟨*medsim.lsp* 24c⟩≡

```
(use-package "WIN32")
(use-package "COM")
⟨asynchronous function call 23c⟩
⟨distributed value receiver 23e⟩
⟨distributed simulator 22f⟩
⟨distributed supervisor 24a⟩
```

This example is of course very simple, but it can be used as the basis for creating a useful framework for managing distributed computations on a network of machines supporting COM.

One issue that I don't yet completely understand is security settings. I ran this example using two NT machines on a local network. To get `CreateInstanceEx` to work for me as an ordinary user (even on the local machine when called with a `:server` argument) I needed to change **Access and Launch** options in `dcomcnfg`'s **Default Security** tab. But more is needed for the receiver callback: without something else

[6] See URL `medsim.lsp`.

I get “Access denied” on the `GetIDsOfNames` call from the worker back to the supervisor when running on a different machine (on the same machine, even when called with `:server` there is no problem). A very dirty workaround is to change the **Default Authentication Level** in the **Default Properties** tab to **(None)** instead of **Connect**. I also ran the supervisor as `Administrator`, but I'm not sure that was necessary.

3 COM Basics

This is basic stuff needed by both servers and clients.

25a ⟨*package specification* 25a⟩≡ (25b 48a)
```
(defpackage "WIN32-COM" (:nicknames "COM") (:use "XLISP"))
(in-package "WIN32-COM")
```

25b ⟨*COM basics* 25b⟩≡ (112a)
```
(let ((major 3)
      (minor 52)
      (subminor 16))
  (unless (or (> system:xls-major-release major)
              (and (= system:xls-major-release major)
                   (> system:xls-minor-release minor))
              (and (= system:xls-major-release major)
                   (= system:xls-minor-release minor)
                   (>= system:xls-subminor-release subminor)))
    (error "COM support requires at least version ~d.~d.~d"
           major minor subminor)))

(require "win32")
(provide "win32com")
⟨package specification 25a⟩

(defvar *com-library*)
(wrap:library-load *com-library*)

(wrap:c-lines "#include \"comutil.h\"")
⟨debugging support 25c⟩
⟨C type declarations 31e⟩
⟨basic constants 109e⟩
⟨error constants 32c⟩
⟨server constants 39d⟩
⟨dispatch constants 42f⟩
⟨type library constants 67d⟩
⟨type information constants 71f⟩
⟨support utilities 108c⟩
⟨initialization and termination 26a⟩
⟨GUID and CLSID functions 27b⟩
⟨locale support 31a⟩
⟨error handling 31d⟩
⟨interfaces 33e⟩
```

3.1 Debugging Support

25c ⟨*debugging support* 25c⟩≡ (25b)

```
(defparameter *com-debug* nil)

(defun debug-print (fmt &rest args)
  (when *com-debug* (apply #'format *debug-io* fmt args)))
```

3.2 Initialization and Termination

26a ⟨*initialization and termination* 26a⟩≡ (25b) 26b ▷

```
(wrap:std-com-function co-initialize "CoInitialize" ((:cptr "void" t)))
(wrap:c-function co-uninitialize "CoUninitialize" () :void)
```

**** use an internal weak hash table here.

26b ⟨*initialization and termination* 26a⟩+≡ (25b) ◁ 26a 26c ▷

```
(defparameter *com-initialized* nil)
(defparameter *com-exit-handlers* nil)
```

26c ⟨*initialization and termination* 26a⟩+≡ (25b) ◁ 26b 26d ▷

```
(defun com-embedding ()
  (if (find "-Embedding" *command-line* :test #'string=) t nil))
```

**** This locks the server process if the embedding flag isn't there. I'm not unlocking it so unloading does not kill the process.

26d ⟨*initialization and termination* 26a⟩+≡ (25b) ◁ 26c 26e ▷

```
(defun uninitialize-com ()
  (when *com-initialized*
    (revoke-all-active-objects)
    (revoke-all-class-factories)
    (disconnect-all-event-listeners)
    (disconnect-all-servers)
    (run-com-exit-handlers)
    (co-uninitialize)
    (setf *com-initialized* nil)
    (setf *com-exit-handlers* nil)))

(defun initialize-com ()
  (unless *com-initialized*
    (co-initialize nil)
    (unless (com-embedding) (add-ref-server-process))
    (system:add-exit-function #'uninitialize-com)
    (setf *com-initialized* t)
    (setf *com-exit-handlers* (make-hash-table))))
```

For now, just put `initialize-com` call in each possible starting point. Once threads are added this might need to be done very differently.

26e ⟨*initialization and termination* 26a⟩+≡ (25b) ◁ 26d 27a ▷

```
(defun register-com-exit-handler (object handler)
  (let ((key (pointer-address (address-of object)))
        (val (cons (make-weak-box object) handler)))
    (setf (gethash key *com-exit-handlers*) val)))

(defun unregister-com-exit-handler (object)
  (let ((key (pointer-address (address-of object))))
    (remhash key *com-exit-handlers*)))
```

```
  (defun run-com-exit-handlers ()
    (unwind-protect
        (maphash #'(lambda (key val)
                     (let ((object (weak-box-value (car val)))
                           (fun (cdr val)))
                       (when object (ignore-errors (funcall fun object)))))
                 *com-exit-handlers*)
      (clrhash *com-exit-handlers*)))
```

27a ⟨*initialization and termination* 26a⟩+≡ (25b) ◁26e

```
  (defun unload-com ()
    (uninitialize-com)
    (let ((lib *com-library*))
      (when lib
        (shlib:close-shared-library lib)
        (setf *com-library* nil))))
```

3.3 GUID and CLSID Functions

27b ⟨GUID *and* CLSID *functions* 27b⟩≡ (25b) 27c▷

```
  (export '(⟨GUID and CLSID exports 27d⟩))
```

3.3.1 GUID Representation

27c ⟨GUID *and* CLSID *functions* 27b⟩+≡ (25b) ◁27b 27e▷

```
  (defstruct (guid (:constructor new-guid (data hash-value))
                   (:print-function (lambda (guid stream d)
                                      (declare (ignore d))
                                      (format stream "#<GUID ~a>"
                                              (guid-string guid)))))
    data string-cache hash-value)
```

27d ⟨GUID *and* CLSID *exports* 27d⟩≡ (27b) 27f▷

```
  guid guid-p
```

27e ⟨GUID *and* CLSID *functions* 27b⟩+≡ (25b) ◁27c 27g▷

```
  (defun guid-string (guid)
    (let ((cache (guid-string-cache guid)))
      (if cache
          cache
        (let ((string (guid-data-to-string (guid-data guid))))
          (setf (guid-string-cache guid) string)
          string))))
```

27f ⟨GUID *and* CLSID *exports* 27d⟩+≡ (27b) ◁27d 29a▷

```
  guid-string
```

27g ⟨GUID *and* CLSID *functions* 27b⟩+≡ (25b) ◁27e 28a▷

```
  (wrap:c-pointer "GUID" (:make make-guid-data))
```

3.3.2 Managing Representation Uniqueness

Makes sure that each GUID has a unique representation so eq can be used for comparison. Allows hashing on guid and the like.

28a ⟨GUID *and* CLSID *functions* 27b⟩+≡ (25b) ◁27g 28c▷

```
(wrap:c-lines "
LVAL GetGuidHash(GUID *pg)
{
  LVAL val, shift, arg;
  unsigned char *p = (unsigned char *) pg;
  int i;

  xlstkcheck(3);
  xlsave(val);
  xlsave(shift);
  xlsave(arg);
  shift = cvfixnum(8);

  val = cvfixnum(p[0]);
  for (i = 1; i < sizeof(GUID); i++) {
    arg = cvfixnum(p[i]);
    val = xlcallsubr2(xash, val, shift);
    val = xlcallsubr2(xadd, val, arg);
  }
  xlpopn(3);
  return val;
}")
(wrap:c-function guid-data-hash-value "GetGuidHash" ((:cptr "GUID")) :lval)
```

;;***** using pointer wrappers:

28b ⟨*Lisp version of* guid-data-hash-value 28b⟩≡

```
(defun guid-data-hash-value (data)
  (let ((ucd (wrapptrs:cast-c-uchar (guid-data g)))
        (val 0))
    (dotimes (i 16 val)
      (setf val (+ (ash val 8) (wrapptrs:get-c-uchar ucd i))))))
```

28c ⟨GUID *and* CLSID *functions* 27b⟩+≡ (25b) ◁28a 28d▷

```
(defvar *guids* (make-hash-table))

(defun make-guid (data)
  (let* ((hash-value (guid-data-hash-value data))
         (guid (gethash hash-value *guids*)))
    (if guid
        guid
        (let ((new-guid (new-guid data hash-value)))
          (setf (gethash hash-value *guids*) new-guid)
          new-guid))))
```

3.3.3 String Representation

**** is this one needed (just use guid-from-string)??

28d ⟨GUID *and* CLSID *functions* 27b⟩+≡ (25b) ◁28c 29b▷
```
(defun clsid-from-string (name)
  (let* ((data (make-guid-data))
         (wname (string-to-wide-string name)))
    (base-clsid-from-string wname data)
    (make-guid data)))

(wrap:std-com-function base-clsid-from-string "CLSIDFromString"
                       ((:cptr "WCHAR") (:cptr "GUID")))
```

29a ⟨GUID *and* CLSID *exports* 27d⟩+≡ (27b) ◁27f 29c▷
```
clsid-from-string
```

29b ⟨GUID *and* CLSID *functions* 27b⟩+≡ (25b) ◁28d 29d▷
```
(defun guid-from-string (name)
  (let* ((data (make-guid-data))
         (wname (string-to-wide-string name)))
    (base-iid-from-string wname data)
    (make-guid data)))

(wrap:std-com-function base-iid-from-string "IIDFromString"
                       ((:cptr "WCHAR") (:cptr "GUID")))
```

29c ⟨GUID *and* CLSID *exports* 27d⟩+≡ (27b) ◁29a 29f▷
```
guid-from-string
```

29d ⟨GUID *and* CLSID *functions* 27b⟩+≡ (25b) ◁29b 29e▷
```
(defun guid-data-to-string (data)
  (let* ((wsize 80)
         (wstring (make-wide-string wsize))
         (res (string-from-guid2 data wstring wsize)))
    (when (= 0 res) (error "buffer to small for GUID conversion"))
    (wide-string-to-string wstring)))

(wrap:c-function string-from-guid2 "StringFromGUID2"
                 ((:cptr "GUID") (:cptr "WCHAR" t) :integer) :integer)
```

3.3.4 ProgID Lookup

29e ⟨GUID *and* CLSID *functions* 27b⟩+≡ (25b) ◁29d 30a▷
```
;;;***** Need to do lookup across the network
(defun clsid-from-progid (name)
  (let* ((clsid (make-guid-data))
         (wname (string-to-wide-string name)))
    (base-clsid-from-progid wname clsid)
    (make-guid clsid)))

(wrap:std-com-function base-clsid-from-progid "CLSIDFromProgID"
                       ((:cptr "WCHAR") (:cptr "GUID")))
```

29f ⟨GUID *and* CLSID *exports* 27d⟩+≡ (27b) ◁29c 30c▷
```
clsid-from-progid
```

30a ⟨GUID *and* CLSID *functions* 27b⟩+≡ (25b) ◁29e 30b▷
```
;;**** not a perfect test but it'll do
(defun guid-string-p (x)
  (and (stringp x) (= (length x) 38) (eql (char x 0) #\{)))

(defun find-clsid (cls-spec)
  (etypecase cls-spec
             (guid cls-spec)
             ((satisfies guid-string-p) (guid-from-string cls-spec))
             (string (clsid-from-progid cls-spec))))
```

3.3.5 Creating New GUIDs

30b ⟨GUID *and* CLSID *functions* 27b⟩+≡ (25b) ◁30a 30d▷
```
(defun create-guid ()
  (let ((guid (make-guid-data)))
    (co-create-guid guid)
    (make-guid guid)))

(wrap:std-com-function co-create-guid "CoCreateGuid" ((:cptr "GUID")))
```

30c ⟨GUID *and* CLSID *exports* 27d⟩+≡ (27b) ◁29f
```
create-guid
```

3.3.6 Standard Interface IDs

30d ⟨GUID *and* CLSID *functions* 27b⟩+≡ (25b) ◁30b 30e▷
```
(defun get-std-iid (name)
  (let ((data (make-guid-data)))
    (unless (base-get-std-iid name data) (error "can't find IID_~a" name))
    (make-guid data)))
```

30e ⟨GUID *and* CLSID *functions* 27b⟩+≡ (25b) ◁30d
```
(wrap:c-lines "
#define CHECK_STD_IID(n,t,p) do { if (strcmp(n,#t)==0) { *p=IID_##t; return TRUE; } } while (0)
static BOOL GetStdIID(char *name, GUID *pguid)
{
  CHECK_STD_IID(name, NULL, pguid);
  CHECK_STD_IID(name, IUnknown, pguid);
  CHECK_STD_IID(name, IDispatch, pguid);
  CHECK_STD_IID(name, ITypeInfo, pguid);
  CHECK_STD_IID(name, ITypeLib, pguid);
  CHECK_STD_IID(name, IEnumVARIANT, pguid);
  CHECK_STD_IID(name, IConnectionPointContainer, pguid);
  CHECK_STD_IID(name, IClassFactory, pguid);
  CHECK_STD_IID(name, IProvideClassInfo, pguid);
  return FALSE;
}
#undef CHECK_STD_IID
")
(wrap:c-function base-get-std-iid "GetStdIID" (:string (:cptr "GUID")) :bool)
```

3.4 Locale Support

31a ⟨*locale support* 31a⟩≡ (25b) 31b ▹
```
(wrap:declare-c-type lcid "LCID" :unsigned)
```

**** export this?

31b ⟨*locale support* 31a⟩+≡ (25b) ◃ 31a 31c ▹
```
(wrap:c-constant LOCALE_SYSTEM_DEFAULT "LOCALE_SYSTEM_DEFAULT" :unsigned)
```

**** get rid of function now that variable works?? **** export variable?

31c ⟨*locale support* 31a⟩+≡ (25b) ◃ 31b
```
(defvar *com-locale* LOCALE_SYSTEM_DEFAULT)
(defun com-locale () *com-locale*)
```

3.5 Error Handling

31d ⟨*error handling* 31d⟩≡ (25b)
```
(export '(⟨error handling exports 32e⟩))
⟨HRESULT decoding 31f⟩
⟨EXCEPINFO decoding 32b⟩
⟨signaling COM errors 32d⟩
```

3.5.1 HRESULT Decoding

31e ⟨*C type declarations* 31e⟩≡ (25b) 38c ▹
```
(wrap:declare-c-type hresult "HRESULT" :unsigned)
```

31f ⟨HRESULT *decoding* 31f⟩≡ (31d) 31g ▹
```
(wrap:c-function hresult-failed "FAILED" (hresult) :bool)
```

31g ⟨HRESULT *decoding* 31f⟩+≡ (31d) ◃ 31f 32a ▹
```
(wrap:c-lines "
static LVAL GetHresultMessage(HRESULT hr)
{
  char *msg = NULL;
  LVAL val;
  DWORD count;

  count = FormatMessage(FORMAT_MESSAGE_ALLOCATE_BUFFER |
                        FORMAT_MESSAGE_FROM_SYSTEM |
                        FORMAT_MESSAGE_IGNORE_INSERTS,
                        NULL, hr, LOCALE_SYSTEM_DEFAULT,
                        (LPTSTR)&msg, 0, NULL);
  val = count > 0 && msg != NULL ? cvstring(msg) : NIL;
  if (msg != NULL) LocalFree(msg);
  return val;
}")

(wrap:c-function base-hresult-to-string "GetHresultMessage" (hresult) :lval)
```

32a ⟨HRESULT *decoding* 31f⟩+≡ (31d) ◁31g

```
(defun hresult-to-string (hr)
  (let ((hrstr (base-hresult-to-string hr)))
    (if hrstr
        (string-right-trim '(#\newline #\return) hrstr)
      "unknown OLE error")))
```

3.5.2 EXCEPINFO Decoding

32b ⟨EXCEPINFO *decoding* 32b⟩≡ (31d)

```
;;**** doesn't run the function pointer if it is there (is it ever??)
(wrap:c-pointer "EXCEPINFO"
                (:get excepinfo-wcode "wCode" :unsigned)
                (:get excepinfo-scode "scode" :unsigned)
                (:get excepinfo-source "bstrSource" (:cptr "WCHAR"))
                (:get excepinfo-description "bstrDescription" (:cptr "WCHAR"))
                (:get excepinfo-help-file "bstrHelpFile" (:cptr "WCHAR"))
                (:get excepinfo-help-context "dwHelpContext" :unsigned))
```

3.5.3 Signaling COM Errors

32c ⟨*error constants* 32c⟩≡ (25b)

```
(wrap:c-constant DISP_E_EXCEPTION "DISP_E_EXCEPTION" :unsigned)
```

32d ⟨*signaling COM errors* 32d⟩≡ (31d) 32f▷

```
(define-condition com-error (error)
  ((hresult :reader com-error-hresult :initarg :hresult)
   (guid :reader com-error-guid :initarg :guid)
   (source :reader com-error-source :initarg :source)
   (description :reader com-error-description :initarg :description)
   (help-file :reader com-error-help-file :initarg :help-file)
   (help-context :reader com-error-help-context :initarg :help-context))
  (:report (lambda (cond stream)
             (let ((hr (com-error-hresult cond))
                   (src (com-error-source cond))
                   (desc (com-error-description cond)))
               (let ((hrstr (if desc desc (hresult-to-string hr))))
                 (format stream "~@[~a: ~]~a" src hrstr))))))
```

32e ⟨*error handling exports* 32e⟩≡ (31d)

```
com-error com-error-guid com-error-hresult
com-error-description com-error-help-file com-error-help-context
```

;;***** need server to fill in Source sensibly

32f ⟨*signaling COM errors* 32d⟩+≡ (31d) ◁32d 33a▷

```
(defun make-idispatch-exception (hr name excep)
  (flet ((bstr2str (ws)
           (when ws (prog1 (wide-string-to-string ws) (free-bstr ws)))))
    (let ((scode (excepinfo-scode excep))
          (source (bstr2str (excepinfo-source excep)))
          (desc (bstr2str (excepinfo-description excep)))
          (help (bstr2str (excepinfo-help-file excep)))
          (helpctxt (excepinfo-help-context excep)))
```

```
        (make-condition 'com-error
                        :hresult (if (hresult-failed scode) scode hr)
                        :source (if source source name)
                        :description (cond
                                      (desc desc)
                                      ((hresult-failed scode)
                                       (hresult-to-string scode))
                                      (t "unknown exception"))
                        :help-file help
                        :help-context helpctxt))))
```

33a ⟨*signaling COM errors* 32d⟩+≡ (31d) ◁32f 33b▷

```
  (defun make-std-com-exception (hr name)
    (let ((hrstr (hresult-to-string hr)))
      (make-condition 'com-error :hresult hr :source name :description hrstr)))
```

**** Should try to use Rich Error Info if in a method call. This means handling method calls and functions separately; method calls need to get the interface that produced the error so thay can call `GetErrorInfo` or whatever it is called.

33b ⟨*signaling COM errors* 32d⟩+≡ (31d) ◁33a 33c▷

```
  (defun raise-com-error (hr &optional name excep)
    (error (if (and excep (= hr DISP_E_EXCEPTION))
               (make-idispatch-exception hr name excep)
             (make-std-com-exception hr name))))
```

33c ⟨*signaling COM errors* 32d⟩+≡ (31d) ◁33b

```
  (wrap:c-callback "RaiseComError" raise-com-error
                   (hresult :string (:cptr "EXCEPINFO")) :void
                   :static nil)
```

33d ⟨*error signaling support declarations* 33d⟩≡ (112b)

```
  void RaiseComError(HRESULT hr, char * fun, EXCEPINFO *e);
```

3.6 Interfaces

33e ⟨*interfaces* 33e⟩≡ (25b)

```
  (export '(⟨interface exports 34a⟩))
  ⟨interface representation 33f⟩
  ⟨finding and registering interfaces 34b⟩
  ⟨interface references 34e⟩
  ⟨declaring interfaces 35c⟩
  ⟨standard interfaces 36b⟩
```

3.6.1 Interface Representation

33f ⟨*interface representation* 33f⟩≡ (33e)

```
  (defstruct (interface
              (:constructor new-interface (name cname constructor caster iid))
              (:print-function (lambda (intf stream d)
                                 (declare (ignore d))
                                 (format stream "#<~a ~a>"
                                         (type-of intf)
                                         (interface-cname intf)))))
    name cname constructor caster iid display)
```

34a ⟨*interface exports* 34a⟩≡ (33e) 34d ▷
```
interface interface-p interface-name interface-cname
interface-iid
```

3.6.2 Finding and Registering Interfaces

34b ⟨*finding and registering interfaces* 34b⟩≡ (33e) 34c ▷
```
(defvar *interfaces* (make-hash-table))

(defun add-interface (interface)
  (let ((name (interface-name interface))
        (iid (interface-iid interface)))
    (when (gethash name *interfaces*)
      (warn "interface ~s is being redefined" name))
    (when (gethash iid *interfaces*)
      (warn "interface with IID ~s is being redefined" iid))
    (setf (gethash name *interfaces*) interface)
    (setf (gethash iid *interfaces*) interface)))
```

**** could also search by cname

34c ⟨*finding and registering interfaces* 34b⟩+≡ (33e) ◁ 34b
```
(defun find-interface (intfspec)
  (etypecase intfspec
             (interface intfspec)
             (guid (gethash intfspec *interfaces*))
             (symbol (gethash intfspec *interfaces*))
             (string (gethash (guid-from-string intfspec) *interfaces*))))

(defun find-interface-iid (intfspec)
  (if (guid-p intfspec)
      intfspec
    (interface-iid (find-interface intfspec))))
```

34d ⟨*interface exports* 34a⟩+≡ (33e) ◁ 34a 36c ▷
```
find-interface find-interface-iid
```

3.6.3 Interface References

**** Is caching of cast pointers worth considering ? **** different name??

34e ⟨*interface references* 34e⟩≡ (33e) 34g ▷
```
(defstruct (com-ref
            (:print-function (lambda (ref stream d)
                               (declare (ignore d))
                               (format stream "#<~s>" (type-of ref)))))
  ⟨com-ref slots 34f⟩)
```

34f ⟨`com-ref` *slots* 34f⟩≡ (34e 35c)
```
address interface-display
```

34g ⟨*interface references* 34e⟩+≡ (33e) ◁34e 35a▷
```
(defun make-com-ref-pointer-accessor (intf)
  (let* ((name (interface-cname intf))
         (type (lookup-pointer-type name))
         (index (- (length (interface-display intf)) 1)))
    (lambda (ref)
      (let ((idisp (com-ref-interface-display ref)))
        (unless (and (< index (length idisp)) (eq (aref idisp index) intf))
          (error "~a is not a reference of type ~a" ref type))
        (base-cast-pointer type (com-ref-address ref))))))
```

**** export this??? **** change name???

35a ⟨*interface references* 34e⟩+≡ (33e) ◁34g 35b▷
```
(defun release-reference (ref)
  (let ((ptr (iunknown-pointer ref)))
    (when ptr
      (disconnect-event-listeners ref)
      (setf (com-ref-address ref) nil)
      (debug-print "Unregistering ... ")
      (unregister-com-exit-handler ref)
      (debug-print "Releasing ~a ... " (type-of ref))
      (base-iunknown-release ptr)
      (debug-print "done~%"))))
```

**** change name???

35b ⟨*interface references* 34e⟩+≡ (33e) ◁35a
```
(defun make-reference (intfspec pointer &rest args)
  (if pointer
      (let ((intf (find-interface intfspec)))
        (unless intf (error "interface ~s is not declared" intfspec))
        (let* ((idisp (interface-display intf))
               (cptr (funcall (interface-caster intf) pointer))
               (ref (apply (interface-constructor intf) cptr idisp args)))
          (system:cptr-protect cptr ref)
          (register-com-exit-handler ref #'release-reference)
          (system:register-finalizer ref #'release-reference)
          ref))
    nil))
```

3.6.4 Declaring Interfaces

**** `*base-interface*` allows using something other than `iunknown` as name. Is this useful?

35c ⟨*declaring interfaces* 35c⟩≡ (33e) 36a▷
```
(defvar *base-interface*)

(defmacro declare-interface (first &rest more)
  (if (consp first)
      `(declare-interface-1 ,first ,@more)
    `(declare-interface-1 (,first) ,@more)))

(defmacro declare-interface-1 ((cname &key name include print iid) &rest slots)
  (let ((name (if name name (intern (string-upcase cname)))))
    (when (string= cname "IUnknown") (setf *base-interface* name))
```

```
      (let* ((constr (intern (concatenate 'string "NEW-" (string name))))
             (parent (cond
                       (include include)
                       ((eq name *base-interface*) 'com-ref)
                       (t *base-interface*)))
             (mods `((:include ,parent)
                     (:constructor ,constr (⟨com-ref slots 34f⟩))))
             (iid-name (intern (concatenate 'string
                                            "IID_" (string-upcase cname)))))
        (when print (push `(:print-function ,print) mods))
        `(progn
           (defstruct (,name ,@mods) ,@slots)
           (add-interface (make-interface ',name ,cname #',constr ,iid ',parent))
           (defconstant ,iid-name (find-interface-iid ',name))
           ',name))))
```

**** separate out definition of pointer accessor function from creation of function (put function in interface as slot, put assignment to symbol into macro)???

36a ⟨*declaring interfaces* 35c⟩+≡ (33e) ◁ 35c

```
  (defun make-interface (name cname constr iid parent)
    (let* ((guid (etypecase iid
                           (null (get-std-iid cname))
                           (string (guid-from-string iid))
                           (guid iid)))
           (cast (make-pointer-caster cname))
           (intf (new-interface name cname constr cast guid))
           (pdisp (if (eq parent 'com-ref)
                      nil
                    (interface-display (find-interface parent))))
           (ptrname (intern (concatenate 'string (string name) "-POINTER"))))
      (setf (interface-display intf) (concatenate 'vector pdisp (list intf)))
      (setf (symbol-function ptrname) (make-com-ref-pointer-accessor intf))
      intf))
```

3.6.5 Some Standard Interfaces

36b ⟨*standard interfaces* 36b⟩≡ (33e) 36d ▷

```
  (declare-interface "IUnknown")
```

36c ⟨*interface exports* 34a⟩+≡ (33e) ◁ 34d 36e ▷

```
  iunknown IID_IUNKNOWN
```

**** Don't declare this as interface since it doesn't make sense and would mess up the null type.

36d ⟨*standard interfaces* 36b⟩+≡ (33e) ◁ 36b 36f ▷

```
  (defconstant IID_NULL (get-std-iid "NULL"))
```

36e ⟨*interface exports* 34a⟩+≡ (33e) ◁ 36c 37a ▷

```
  IID_NULL
```

36f ⟨*standard interfaces* 36b⟩+≡ (33e) ◁ 36d 37b ▷

```
  (declare-interface ("IDispatch"
                      :print (lambda (ref stream d)
                               (let ((info-name (idispatch-info-name ref))
                                     (type (type-of ref)))
                                 (if info-name
```

```
                                    (format stream "#<~s ~a>" type info-name)
                                  (format stream "#<~s>" type)))))
    (info-name-cache :empty))

  (defun idispatch-info-name (ref)
    (let ((cache (idispatch-info-name-cache ref)))
      (if (eq cache :empty)
          (let* ((info (idispatch-type-info ref))
                 (name (when info (itypeinfo-name info))))
            (setf (idispatch-info-name-cache ref) name)
            name)
        cache)))
```

37a ⟨*interface exports* 34a⟩+≡ (33e) ◁36e 37c▷

```
  idispatch IID_IDISPATCH
```

37b ⟨*standard interfaces* 36b⟩+≡ (33e) ◁36f 37d▷

```
  (declare-interface ("ITypeInfo"
                      :print (lambda (ref stream d)
                               (declare (ignore d))
                               (let ((type (type-of ref))
                                     (name (itypeinfo-name ref)))
                                 (format stream "#<~a~@[ ~a~]>" type name)))))

  ;;**** avoid lookup??
  (defun wrap-itypeinfo (ptr)
    (make-reference 'itypeinfo ptr))
```

37c ⟨*interface exports* 34a⟩+≡ (33e) ◁37a 37e▷

```
  itypeinfo IID_ITypeInfo
```

37d ⟨*standard interfaces* 36b⟩+≡ (33e) ◁37b 37f▷

```
  (declare-interface ("ITypeLib"
                      :print (lambda (ref stream d)
                               (declare (ignore d))
                               (let ((type (type-of ref))
                                     (name (itypelib-name ref)))
                                 (format stream "#<~a~@[ ~a~]>" type name)))))

  ;;**** avoid lookup??
  (defun wrap-itypelib (ptr)
    (make-reference 'itypelib ptr))
```

37e ⟨*interface exports* 34a⟩+≡ (33e) ◁37c

```
  itypelib IID_ITypeLib
```

**** flesh these out, add exports; maybe add a few more.

37f ⟨*standard interfaces* 36b⟩+≡ (33e) ◁37d

```
  (declare-interface "IClassFactory")
  (declare-interface "IConnectionPointContainer")
  (declare-interface "IEnumVARIANT")
```

4 Client Support

38a ⟨*COM client support* 38a⟩≡ (112a)
```
(export '(⟨client support exports 39b⟩))
⟨creating object references 38b⟩
⟨IUnknown interface 41d⟩
⟨IDispatch interface 42c⟩
⟨automation support 46b⟩
⟨variant type conversion 47d⟩
```

4.1 Creating Object References

38b ⟨*creating object references* 38b⟩≡ (38a)
```
⟨creating new instances 38d⟩
⟨binding to objects 39e⟩
⟨accessing active objects 40c⟩
⟨returning object references 40e⟩
```

4.1.1 Creating New Instances

38c ⟨*C type declarations* 31e⟩+≡ (25b) ◁31e 42a▷
```
(wrap:declare-c-type dword "DWORD" :unsigned)
```

38d ⟨*creating new instances* 38d⟩≡ (38b) 39a▷
```
(wrap:std-com-function co-create-instance "CoCreateInstance"
                        ((:cptr "GUID") (:cptr "IUnknown" t)
                         dword (:cptr "GUID")
                         (:value (:cptr "void"))))

(wrap:c-lines "
static HRESULT CreateRemoteInstance(CLSID *cid, IUnknown *pouter, DWORD cntxt,
                                    WCHAR *server, IID *iid, void **ppunk)
{
  HRESULT hr;
  MULTI_QI mqi;
  COSERVERINFO csi;

  if (pCoCreateInstanceEx == NULL) xlfail(\"DCOM not available\");

  memset(&mqi, 0, sizeof(MULTI_QI));
  memset(&csi, 0, sizeof(COSERVERINFO));
  mqi.pIID = iid;
  csi.pwszName = server;
  *ppunk = NULL;
  hr = pCoCreateInstanceEx(cid, pouter, cntxt, &csi, 1, &mqi);
  *ppunk = mqi.pItf;
  return FAILED(hr) ? hr : mqi.hr;
}")

(wrap:std-com-function create-remote-instance "CreateRemoteInstance"
                  ((:cptr "GUID") (:cptr "IUnknown" t) dword
                   (:cptr "WCHAR") (:cptr "GUID")
                   (:value (:cptr "void"))))
```

**** test remote version!! **** Why does DAO.DBEngine.35 return an Unknown OLE Error? **** For DLL components, do I need to check for CanUnload??

39a ⟨*creating new instances* 38d⟩+≡ (38b) ◁38d

```
(defun base-create-object (clsid outer cntxt server)
  (let ((iid (guid-data IID_IUnknown)))
    (system:without-interrupts
     (make-reference
      IID_IUnknown
      (if server
          (let ((wserver (string-to-wide-string server)))
            (create-remote-instance clsid outer cntxt wserver iid))
        (co-create-instance clsid outer cntxt iid))))))

(defun create-object (cls-spec &key
                               outer
                               (context :server)
                               (interface 'idispatch)
                               server)
  (initialize-com)
  (let* ((clsid (guid-data (find-clsid cls-spec)))
         (pouter (if outer (iunknown-pointer outer) nil))
         (cntxt ⟨translate keyword to CLSCTX value 39c⟩)
         (unk (base-create-object clsid pouter cntxt server)))
    (query-interface unk interface)))
```

39b ⟨*client support exports* 39b⟩≡ (38a) 40b▷

```
create-object
```

**** move this to the right place The registration function is given the context to use as a keyword. The possible keywords are `:inproc`, `:local` and `:server`; these are translated to the C level as

39c ⟨*translate keyword to* `CLSCTX` *value* 39c⟩≡ (39a 101e)

```
(ecase context
       (:inproc CLSCTX_INPROC_SERVER)
       (:local (logior CLSCTX_INPROC_SERVER
                       CLSCTX_LOCAL_SERVER))
       (:server CLSCTX_SERVER))
```

Several constants are used to specify the registration context.

39d ⟨*server constants* 39d⟩≡ (25b)

```
(wrap:c-constant CLSCTX_INPROC_SERVER "CLSCTX_INPROC_SERVER" :unsigned)
(wrap:c-constant CLSCTX_LOCAL_SERVER "CLSCTX_LOCAL_SERVER" :unsigned)
(wrap:c-constant CLSCTX_SERVER "CLSCTX_SERVER" :unsigned)
```

4.1.2 Binding To Objects

**** allow "to left" moniker?? **** higher level moniker access??

39e ⟨*binding to objects* 39e⟩≡ (38b) 40a▷

```
(wrap:c-lines "
static HRESULT MyGetObject(WCHAR *name, REFIID iid, void **ppval)
{
  HRESULT hr;
  IBindCtx *pbind = NULL;
  IMoniker *pmon = NULL;
```

```
  DWORD len;

  hr = CreateBindCtx(0, &pbind);
  if (SUCCEEDED(hr)) {
    hr = MkParseDisplayName(pbind, name, &len, &pmon);
    if (SUCCEEDED(hr)) {
      hr = IMoniker_BindToObject(pmon, pbind, NULL, iid, ppval);
      IUnknown_Release(pmon);
    }
    IUnknown_Release(pbind);
  }
  return hr;
}")
(wrap:std-com-function base-get-object "MyGetObject"
                       ((:cptr "WCHAR") (:cptr "GUID")
                        (:value (:cptr "void"))))
```

40a ⟨*binding to objects* 39e⟩+≡ (38b) ◁39e

```
(defun get-object (name &optional (interface 'idispatch))
  (initialize-com)
  (let ((iid-data (guid-data (find-interface-iid interface)))
        (wname (string-to-wide-string name)))
    (system:without-interrupts
     (make-reference interface (base-get-object wname iid-data)))))
```

40b ⟨*client support exports* 39b⟩+≡ (38a) ◁39b 40d▷

```
get-object
```

4.1.3 Accessing Active Objects

**** fix this use of query interface stuff??

40c ⟨*accessing active objects* 40c⟩≡ (38b)

```
(wrap:std-com-function base-get-active-object "GetActiveObject"
                       ((:cptr "GUID") (:cptr "void" t)
                        (:value (:cptr "IUnknown"))))

(defun get-active-object (cls-spec &optional (interface 'idispatch))
  (initialize-com)
  (let* ((clsid (guid-data (find-clsid cls-spec)))
         (unk (system:without-interrupts
                (make-reference 'iunknown (base-get-active-object clsid nil)))))
    (query-interface unk interface)))
```

40d ⟨*client support exports* 39b⟩+≡ (38a) ◁40b 42b▷

```
get-active-object
```

4.1.4 Returning Object References

40e ⟨*returning object references* 40e⟩≡ (38b) 41b▷

```
;;**** trap errors, return NULL?
;;**** should this be released before close?
(wrap:c-callback "IDispatch2Lisp" wrap-idispatch ((:cptr "IDispatch")) :lval
                 :static nil
```

```
                 :interrupts-allow nil)

;;;**** avoid the lookup here??
(defun wrap-idispatch (ref)
  (make-reference 'idispatch ref))
```

41a ⟨*type conversion declarations* 41a⟩≡ (112b) 41c ▷

```
LVAL IDispatch2Lisp(IDispatch *pdisp);
```

**** Issue: exit can happen inside unwind-protect–ought to throw out of current context to exit point.
**** move someplace else?

41b ⟨*returning object references* 40e⟩+≡ (38b) ◁ 40e

```
;;**** avoid lookup??
(defun wrap-iunknown (ref)
  (make-reference 'iunknown ref))

(wrap:c-callback "IUnknown2Lisp" wrap-iunknown ((:cptr "IUnknown")) :lval
                 :static nil
                 :interrupts-allow nil)
```

41c ⟨*type conversion declarations* 41a⟩+≡ (112b) ◁ 41a 52c ▷

```
LVAL IUnknown2Lisp(IUnknown *punk);
```

4.2 The IUnknown Interface

41d ⟨IUnknown *interface* 41d⟩≡ (38a) 41e ▷

```
(wrap:c-lines "
static HRESULT BaseQueryInterface(IUnknown *punk, REFIID riid, void **val,
                                  BOOL not_found_is_error)
{
  HRESULT hr = IUnknown_QueryInterface(punk, riid, val);
  if (hr ==  E_NOINTERFACE && ! not_found_is_error) {
    *val = NULL;
    return S_OK;
  }
  else return hr;
}")

(wrap:std-com-function base-query-interface "BaseQueryInterface"
                       ((:cptr "IUnknown") (:cptr "GUID")
                        (:value (:cptr "void")) bool))
#|
(wrap:c-function base-iunknown-addref "IUnknown_AddRef"
                 ((:cptr "IUnknown")) ulong)
|#

(wrap:c-function base-iunknown-release "IUnknown_Release"
                 ((:cptr "IUnknown")) ulong)
```

41e ⟨IUnknown *interface* 41d⟩+≡ (38a) ◁ 41d

```
(defun query-interface (ref iidspec &optional (not-found nil nfsupp))
  (system:without-interrupts
   (let* ((ptr (iunknown-pointer ref))
          (iid (find-interface-iid iidspec))
```

```
          (iid-data (guid-data iid))
          (nptr (base-query-interface ptr iid-data (not nfsupp))))
     (if nptr (make-reference iid nptr) not-found))))
```

42a ⟨*C type declarations* 31e⟩+≡ (25b) ◁38c 42g▷

```
(wrap:declare-c-type bool "BOOL" :bool)
(wrap:declare-c-type ulong "ULONG" :unsigned)
```

42b ⟨*client support exports* 39b⟩+≡ (38a) ◁40d 42e▷

```
query-interface release-reference
```

4.3 The IDispatch Interface

42c ⟨`IDispatch` *interface* 42c⟩≡ (38a)
  ⟨`IDispatch` *type information* 42d⟩
  ⟨*dispatch IDs* 43a⟩
  ⟨*method invokation* 44d⟩

4.3.1 Type Information

Rename to dispatch-type-info???

42d ⟨`IDispatch` *type information* 42d⟩≡ (42c)

```
(wrap:std-com-function base-idispatch-get-type-info-count
                       "IDispatch_GetTypeInfoCount"
                       ((:cptr "IDispatch") (:value dword)))

(wrap:std-com-function base-idispatch-get-type-info "IDispatch_GetTypeInfo"
                       ((:cptr "IDispatch") :unsigned lcid
                        (:value (:cptr "ITypeInfo"))))

(defun idispatch-type-info (ref &optional
                                (index 0) (lcid LOCALE_SYSTEM_DEFAULT))
  (let ((pdisp (idispatch-pointer ref)))
    (values
     (ignore-errors
      (system:without-interrupts
       (when (/= 0 (base-idispatch-get-type-info-count pdisp))
         (make-reference 'itypeinfo
                         (base-idispatch-get-type-info pdisp index lcid)))))))))
```

42e ⟨*client support exports* 39b⟩+≡ (38a) ◁42b 44a▷

```
idispatch-type-info
```

4.3.2 Dispatch IDs

42f ⟨*dispatch constants* 42f⟩≡ (25b) 44b▷

```
(wrap:c-constant DISPID_VALUE "DISPID_VALUE" :unsigned)
```

42g ⟨*C type declarations* 31e⟩+≡ (25b) ◁42a 44c▷

```
(wrap:declare-c-type dispid "DISPID" :unsigned)
```

43a ⟨*dispatch IDs* 43a⟩≡ (42c) 43b ▷
```
(wrap:c-pointer "DISPID"
                (:make make-dispid)
                (:get get-dispid nil dispid)
                (:set set-dispid nil dispid))
```

43b ⟨*dispatch IDs* 43a⟩+≡ (42c) ◁ 43a 43c ▷
```
(wrap:c-pointer (:cptr "WCHAR")
                (:make make-wide-string-array)
                (:get get-cptr-wchar nil (:cptr "WCHAR")))

(wrap:c-lines "
static void set_cptr_wchar(WCHAR **pw, WCHAR *w, int off) {
  pw[off] = w;
}")
(wrap:c-function set-cptr-wchar "set_cptr_wchar"
                 ((:cptr (:cptr "WCHAR")) (:cptr "WCHAR") :integer) :void)
```

`cptr-protect` pushes each new wide string onto the array's protection list.

43c ⟨*dispatch IDs* 43a⟩+≡ (42c) ◁ 43b 43d ▷
```
(defun names-to-wide-string-array (names)
  (let ((wnames (make-wide-string-array (length names)))
        (i 0))
    (dolist (s names wnames)
      (let ((ws (string-to-wide-string (string s))))
        (system:cptr-protect wnames ws)
        (set-cptr-wchar wnames ws i)
        (incf i)))))
```

43d ⟨*dispatch IDs* 43a⟩+≡ (42c) ◁ 43c 43e ▷
```
(defun dispids-to-list (dispids n)
  (let ((val nil))
    (dotimes (i n (nreverse val))
      (push (get-dispid dispids i) val))))
```

43e ⟨*dispatch IDs* 43a⟩+≡ (42c) ◁ 43d 43f ▷
```
(wrap:std-com-function base-idispatch-ids-of-names "IDispatch_GetIDsOfNames"
                       ((:cptr "IDispatch") (:cptr "GUID")
                        (:cptr (:cptr "WCHAR")) :unsigned lcid
                        (:cptr "DISPID")))
```

**** need to do some caching here??

43f ⟨*dispatch IDs* 43a⟩+≡ (42c) ◁ 43e
```
(defun idispatch-ids-of-names (object name &optional keys)
  (if name
      (let* ((names (cons name keys))
             (n (length names))
             (wnames (names-to-wide-string-array names))
             (dispids (make-dispid n))
             (disp (idispatch-pointer object))
             (locale (com-locale))
             (iid-data (guid-data IID_NULL)))
        (base-idispatch-ids-of-names disp iid-data wnames n locale dispids)
        (let ((val (dispids-to-list dispids n)))
```

```
            (values (first val) (rest val))))
      (if keys
          (error "can't have named arguments with default mehtod")
        DISPID_VALUE)))
```

44a ⟨*client support exports* 39b⟩+≡ (38a) ◁42e 44f▷

```
  idispatch-ids-of-names
```

4.3.3 Method Invokation

44b ⟨*dispatch constants* 42f⟩+≡ (25b) ◁42f 51a▷

```
  (wrap:c-constant DISPID_PROPERTYPUT "DISPID_PROPERTYPUT" :unsigned)
  (wrap:c-constant DISPATCH_PROPERTYPUT "DISPATCH_PROPERTYPUT" :unsigned)
  (wrap:c-constant DISPATCH_PROPERTYPUTREF "DISPATCH_PROPERTYPUTREF" :unsigned)
  (wrap:c-constant DISPATCH_METHOD "DISPATCH_METHOD" :unsigned)
  (wrap:c-constant DISPATCH_PROPERTYGET "DISPATCH_PROPERTYGET" :unsigned)
```

44c ⟨*C type declarations* 31e⟩+≡ (25b) ◁42g 50d▷

```
  (wrap:declare-c-type word "WORD" :unsigned)
```

44d ⟨*method invokation* 44d⟩≡ (42c) 44e▷

```
  (wrap:c-function base-invoke "InvokeMethod"
          ((:cptr "IDispatch") dispid lcid word bool :lval :lval)
          :lval)
```

**** allow dispid to be a symbol/string?? **** allow keys to be symbols/strings??

44e ⟨*method invokation* 44d⟩+≡ (42c) ◁44d

```
  (defun idispatch-invoke (ref dispid type &optional
                               args
                               (keydispids
                                (if (member type '(:propput :propputref))
                                    (list DISPID_PROPERTYPUT)
                                  nil))
                               (for-value
                                (if (member type '(:propput :propputref))
                                    nil
                                  t))
                               (locale (com-locale)))
    (let ((pdisp (idispatch-pointer ref))
          (flags (ecase type
                        (:method (logior DISPATCH_METHOD DISPATCH_PROPERTYGET))
                        (:propget (logior DISPATCH_METHOD DISPATCH_PROPERTYGET))
                        (:propput DISPATCH_PROPERTYPUT)
                        (:propputref DISPATCH_PROPERTYPUTREF))))
      (base-invoke pdisp dispid locale flags for-value args keydispids)))
```

44f ⟨*client support exports* 39b⟩+≡ (38a) ◁44a 46c▷

```
  idispatch-invoke
```

44g ⟨*invokation support functions* 44g⟩≡ (112c)

```
  /**** move allocation to Lisp?? avoids danngling pointers on error. */
  LVAL InvokeMethod(IDispatch *pdisp, DISPID id, LCID locale, WORD wFlags,
                    BOOL forValue, LVAL args, LVAL nargs)
  {
    HRESULT hr;
```

```
    UINT argErr;
    EXCEPINFO excepinfo;
    VARIANT result;
    DISPPARAMS dispParams;
    DISPID *nargv;
    int argc, nargc, i;
    VARIANTARG *argv;
    LVAL next;

    /* create the argument structure */
    argc = llength(args);
    if (argc > 0) {
      argv = calloc(argc, sizeof(VARIANTARG));
      if (argv == NULL) xlfail("argument allocation failed");
    }
    else argv = NULL;
    nargc = llength(nargs);
    if (nargc > 0) {
      nargv = calloc(nargc, sizeof(DISPID));
      if (nargv == NULL) xlfail("named argument allocation failed");
    }
    else nargv = NULL;
    for (i = 0, next = args; i < argc; i++, next = cdr(next))
      Lisp2Variant(car(next), VT_VARIANT, &argv[argc - i - 1], TRUE);
    for (i = 0, next = nargs; i < nargc; i++, next = cdr(next))
      nargv[nargc - i - 1] = lisp2ulong(car(next));
    memset(&dispParams, 0, sizeof(DISPPARAMS));
    dispParams.rgvarg = argv;
    dispParams.rgdispidNamedArgs = nargv;
    dispParams.cNamedArgs = nargc;
    dispParams.cArgs = argc;

    /* initialize the result and excetion info */
    if (forValue) VariantInit(&result);
    memset(&excepinfo, 0, sizeof(EXCEPINFO));

    /* invoke the method */
    /**** Perl kludge for WORD? */
    hr = IDispatch_Invoke(pdisp, id, &IID_NULL, locale, wFlags, &dispParams,
                          forValue ? &result : NULL, &excepinfo, &argErr);

    /* clean up and return */
    for (i = 0; i < argc; i++)
      VariantClear(&argv[i]);
    if (argv != NULL) free(argv);
    if (nargv != NULL) free(nargv);
    if (FAILED(hr)) {
      if (forValue) VariantClear(&result);
      RaiseComError(hr, "Invoke", &excepinfo);
    }
    {
      LVAL val;
      BEGIN_PROTECT {
```

```
      val = forValue ? Variant2Lisp(&result) : NIL;
    }
    BEGIN_CLEANUP {
      if (forValue) VariantClear(&result);
    }
    END_PROTECT
    return val;
  }
}
```

46a ⟨*invokation support declarations* 46a⟩≡ (112b)

```
LVAL InvokeMethod(IDispatch *pdisp, DISPID id, LCID locale, USHORT wFlags,
                  BOOL forValue, LVAL args, LVAL nargs);
```

4.4 Automation Support

46b ⟨*automation support* 46b⟩≡ (38a) 46d ▷

```
(defun property (ref &optional meth &rest args)
  (let ((prop-id (if (numberp meth) meth (idispatch-ids-of-names ref meth))))
    (idispatch-invoke ref prop-id :propget args)))
```

46c ⟨*client support exports* 39b⟩+≡ (38a) ◁ 44f 46e ▷

```
property
```

46d ⟨*automation support* 46b⟩+≡ (38a) ◁ 46b 46f ▷

```
(defun put-property (ref meth &rest args)
  (let ((prop-id (if (numberp meth) meth (idispatch-ids-of-names ref meth))))
    (idispatch-invoke ref prop-id :propput args)))
(defsetf property put-property)
```

46e ⟨*client support exports* 39b⟩+≡ (38a) ◁ 46c 46g ▷

```
put-property
```

**** Perl uses REF when value is an object–is that a reasonable heuristic? That seems to hold for VBS; the one case of a REF in Excell seems weird.

46f ⟨*automation support* 46b⟩+≡ (38a) ◁ 46d 46h ▷

```
(defun put-property-ref (ref meth &rest args)
  (let ((prop-id (if (numberp meth) meth (idispatch-ids-of-names ref meth))))
    (idispatch-invoke ref prop-id :propputref args)))
(defsetf property-ref put-property-ref)
```

46g ⟨*client support exports* 39b⟩+≡ (38a) ◁ 46e 47b ▷

```
put-property-ref
```

46h ⟨*automation support* 46b⟩+≡ (38a) ◁ 46f 47a ▷

```
(defun split-invoke-keys (keys)
  (let ((keywords nil)
        (keyargs nil))
    (unless (evenp (length keys)) (error "bad keyword argument list"))
    (loop
     (unless keys (return (values (nreverse keywords) (nreverse keyargs))))
     (let ((k (pop keys))
           (a (pop keys)))
       (unless (keywordp k) (error "~s is not a keyword" k))
       (push k keywords)
       (push a keyargs)))))
```

47a ⟨*automation support* 46b⟩+≡ (38a) ◁46h
```
(defun invoke-1 (object meth for-value args)
  (let ((keys (member-if #'keywordp args)))
    (if keys
        (let ((baseargs (ldiff args keys)))
          (multiple-value-bind (keywords keyargs) (split-invoke-keys keys)
            (multiple-value-bind (meth-id key-ids)
              (idispatch-ids-of-names object meth keywords)
              (let ((args (append baseargs keyargs)))
                (idispatch-invoke object meth-id :method
                                  args key-ids for-value)))))
      (let ((meth-id (if (numberp meth)
                         meth
                       (idispatch-ids-of-names object meth))))
        (idispatch-invoke object meth-id :method args nil for-value)))))

(defun invoke (object meth &rest args)
  (invoke-1 object meth t args))

(defun invoke-no-value (object meth &rest args)
  (invoke-1 object meth nil args))
```

47b ⟨*client support exports* 39b⟩+≡ (38a) ◁46g
```
invoke invoke-no-value
```

47c ⟨*test* 47c⟩≡ 63c ▷
```
;;(setf x (get-object "e:\\my documents\\fred.xls"))
;;(setf s (property x :worksheets 1))
(setf x (get-active-object "Excel.application"))
(setf s (property (property x :workbooks 1) :worksheets 1))
(setf r (property s :range "A1" "C2"))
(property r :value)
(property (property s :cells 1 2) :value)
(invoke x :quit)

(setf ie (create-object "InternetExplorer.application"))
(invoke ie :navigate :flags 1 :url "www.stat.umn.edu")
```

**** mention missing thingy, null thingy here.

4.5 Variant Type Conversion

47d ⟨*variant type conversion* 47d⟩≡ (38a)

(export '(⟨*variant exports* 48d⟩))
⟨*variant type constants* 48b⟩
⟨*variant structure interface* 48c⟩
⟨*currency conversion* 51e⟩
⟨*structure and object conversion* 52d⟩
⟨*Lisp to variant conversion* 59b⟩
⟨*variant to Lisp conversion* 61b⟩
⟨*variant collections* 62d⟩

4.5.1 Variant Type Constants

48a ⟨*commacro.lsp* 48a⟩≡
```
⟨package specification 25a⟩

(defmacro declare-variant-type (name)
  `(progn (export ',name)
          (wrap:c-constant ,name ,(string name) :unsigned)))
```

48b ⟨*variant type constants* 48b⟩≡ (47d)
```
(declare-variant-type VT_ARRAY)
(declare-variant-type VT_BYREF)
(declare-variant-type VT_EMPTY)
(declare-variant-type VT_NULL)
(declare-variant-type VT_UI1)
(declare-variant-type VT_I2)
(declare-variant-type VT_I4)
(declare-variant-type VT_R4)
(declare-variant-type VT_R8)
(declare-variant-type VT_CY)
(declare-variant-type VT_BSTR)
(declare-variant-type VT_ERROR)
(declare-variant-type VT_BOOL)
(declare-variant-type VT_DATE)
(declare-variant-type VT_DISPATCH)
(declare-variant-type VT_VARIANT)
(declare-variant-type VT_UNKNOWN)
```

4.5.2 Lisp-Level Variant Representation

48c ⟨*variant structure interface* 48c⟩≡ (47d) 48e ▷
```
(defstruct (variant (:constructor base-make-variant (pointer))
                    (:print-function print-variant))
  pointer)
```

48d ⟨*variant exports* 48d⟩≡ (47d) 49e ▷
```
variant
```

48e ⟨*variant structure interface* 48c⟩+≡ (47d) ◁ 48c 48f ▷
```
(defun print-variant (var stream d)
  (let* ((ptr (variant-pointer var))
         (type (when ptr (base-variant-type ptr))))
    (format stream "#<~a~@[, type = ~a~]>"
            (type-of var)
            (when type (variant-type-to-string type)))))
```

48f ⟨*variant structure interface* 48c⟩+≡ (47d) ◁ 48e 49a ▷
```
(defun variant-type-to-string (type)
  (let ((byref (/= (logand type VT_BYREF) 0))
        (array (/= (logand type VT_ARRAY) 0))
        (base-type (logand type (lognot (logior VT_BYREF VT_ARRAY)))))
    (format nil "~@[~*VT_BYREF|~]~@[~*VT_ARRAY|~]~a"
            byref
            array
```

```
            (case base-type
                  (0 "VT_EMPTY")
                  (1 "VT_NULL")
                  (2 "VT_I2")
                  (3 "VT_I4")
                  (4 "VT_R4")
                  (5 "VT_R8")
                  (6 "VT_CY")
                  (7 "VT_DATE")
                  (8 "VT_BSTR")
                  (9 "VT_DISPATCH")
                  (10 "VT_ERROR")
                  (11 "VT_BOOL")
                  (12 "VT_VARIANT")
                  (13 "VT_UNKNOWN")
                  (17 "VT_UI1")
                  (t base-type)))))
```

49a ⟨*variant structure interface* 48c⟩+≡ (47d) ◁48f 49b▷

```
;;********* fix wrapper
(defun release-variant (var)
  (let ((ptr (variant-pointer var)))
    (when ptr
      (setf (variant-pointer var) nil)
      (debug-print "Unregistering ... ")
      (unregister-com-exit-handler var)
      (debug-print "Releasing VARIANT ... ")
      (variant-clear ptr)
      (debug-print "done~%"))))

(defun wrap-variant (ptr)
  (let ((var (base-make-variant ptr)))
    (register-com-exit-handler var #'release-variant)
    (system:register-finalizer var #'release-variant)
    var))
```

49b ⟨*variant structure interface* 48c⟩+≡ (47d) ◁49a 49c▷

```
(wrap:c-pointer "VARIANTARG" (:make new-variant) (:offset offset-variant))
```

49c ⟨*variant structure interface* 48c⟩+≡ (47d) ◁49b 49d▷

```
(wrap:std-com-function variant-clear "VariantClear" ((:cptr "VARIANTARG")))
```

4.5.3 Higher Level Interface

49d ⟨*variant structure interface* 48c⟩+≡ (47d) ◁49c 50a▷

```
(defun make-variant (arg &optional (type VT_VARIANT))
  (initialize-com)
  (let ((var (new-variant)))
    (lisp-to-variant arg type var nil)
    (wrap-variant var)))
```

49e ⟨*variant exports* 48d⟩+≡ (47d) ◁48d 50b▷

```
make-variant
```

50a ⟨*variant structure interface* 48c⟩+≡ (47d) ◁49d 50c▷
```
(defun variant-type (var)
  (let ((ptr (variant-pointer var)))
    (when ptr (base-variant-type ptr))))
```

50b ⟨*variant exports* 48d⟩+≡ (47d) ◁49e 50f▷
```
variant-type
```

50c ⟨*variant structure interface* 48c⟩+≡ (47d) ◁50a 50e▷
```
(wrap:c-function base-variant-type "V_VT" ((:cptr "VARIANTARG")) vartype)
```

50d ⟨*C type declarations* 31e⟩+≡ (25b) ◁44c 68g▷
```
(wrap:declare-c-type vartype "VARTYPE" :unsigned)
```

50e ⟨*variant structure interface* 48c⟩+≡ (47d) ◁50c 50g▷
```
(defun variant-value (var)
  (variant-to-lisp (variant-pointer var)))
```

50f ⟨*variant exports* 48d⟩+≡ (47d) ◁50b 50h▷
```
variant-value
```

50g ⟨*variant structure interface* 48c⟩+≡ (47d) ◁50e 50i▷
```
(defun variant-change-type (var type)
  (unless (= (logand type VT_BYREF) 0)
    (error "VT_BYREF flag is not allowed"))
  (let ((new-var (new-variant))
        (ptr (variant-pointer var))
        (lcid (com-locale)))
    (variant-init new-var)
    (base-variant-change-type-ex new-var ptr lcid 0 type)
    (wrap-variant new-var)))
```

50h ⟨*variant exports* 48d⟩+≡ (47d) ◁50f 50k▷
```
variant-change-type
```

50i ⟨*variant structure interface* 48c⟩+≡ (47d) ◁50g 50j▷
```
(wrap:c-function variant-init "VariantInit" ((:cptr "VARIANTARG")) :void)
```

50j ⟨*variant structure interface* 48c⟩+≡ (47d) ◁50i 50l▷
```
(defun variant-value-as (var type)
  (let ((new-var (new-variant))
        (ptr (variant-pointer var))
        (lcid (com-locale)))
    (variant-init new-var)
    (unwind-protect
        (progn
          (base-variant-change-type-ex new-var ptr lcid 0 type)
          (variant-to-lisp new-var))
      (variant-clear new-var))))
```

50k ⟨*variant exports* 48d⟩+≡ (47d) ◁50h 51c▷
```
variant-value-as
```

50l ⟨*variant structure interface* 48c⟩+≡ (47d) ◁50j 51b▷
```
(wrap:std-com-function base-variant-change-type-ex "VariantChangeTypeEx"
                       ((:cptr "VARIANTARG") (:cptr "VARIANTARG")
                        lcid :unsigned vartype))
```

51a ⟨*dispatch constants* 42f⟩+≡ (25b) ◁44b 62c▷
```
(wrap:c-constant DISP_E_PARAMNOTFOUND "DISP_E_PARAMNOTFOUND" :unsigned)
```

51b ⟨*variant structure interface* 48c⟩+≡ (47d) ◁50l
```
(defun make-missing-variant () (make-variant DISP_E_PARAMNOTFOUND VT_ERROR))
(defun make-empty-variant () (make-variant nil VT_EMPTY))
(defun make-null-variant () (make-variant nil VT_NULL))
```

51c ⟨*variant exports* 48d⟩+≡ (47d) ◁50k 60d▷
```
make-missing-variant
make-empty-variant
make-null-variant
```

4.5.4 Currency Conversion

51d ⟨*variant type conversion declarations* 51d⟩≡ (112b) 52e▷
```
LVAL CY2Lisp(ULONG hi, ULONG lo);
void Lisp2CY(LVAL val, CY *pcy);
```

51e ⟨*currency conversion* 51e⟩≡ (47d) 51f▷
```
(wrap:c-callback "CY2Lisp" currency-to-lisp (:unsigned :unsigned) :lval
                 :static nil)
(wrap:c-callback "Lisp2CY" lisp-to-currency (:lval (:cptr "CY")) :void
                 :static nil)
```

51f ⟨*currency conversion* 51e⟩+≡ (47d) ◁51e 51g▷
```
(defconstant currency-cut (expt 2 32))
(defconstant min-currency (- (expt 2 63)))
(defconstant max-currency (- (expt 2 63) 1))
```

51g ⟨*currency conversion* 51e⟩+≡ (47d) ◁51f 51h▷
```
(defun currency-to-lisp (hi lo)
  (let* ((neg (not (zerop (logand (expt 2 31) hi))))
         (ahi (if neg (- currency-cut hi 1) hi))
         (alo (if neg (- currency-cut lo) lo))
         (aval (xlisp:divide (+ (* currency-cut ahi) alo) 10000)))
    (if neg (- aval) aval)))
```

51h ⟨*currency conversion* 51e⟩+≡ (47d) ◁51g 51i▷
```
(defun lisp-to-currency (val pcy)
  (let ((ival (round (* val 10000))))
    (unless (<= min-currency ival max-currency)
      (error "the value ~a is out of the currency range" ival))
    (multiple-value-bind (ahi alo) (floor (abs ival) currency-cut)
      (let ((hi (if (minusp ival) (- currency-cut ahi 1) ahi))
            (lo (if (minusp ival) (- currency-cut alo) alo)))
        (set-currency pcy hi lo)))))
```

51i ⟨*currency conversion* 51e⟩+≡ (47d) ◁51h
```
(wrap:c-lines "
static void SetCY(CY *pcy, ULONG hi, ULONG lo)
{
  pcy->s.Hi = (long) hi;
  pcy->s.Lo = lo;
}")
(wrap:c-function set-currency "SetCY" ((:cptr "CY") :unsigned :unsigned) :void)
```

4.5.5 String Conversion

52a ⟨*variant type conversion functions* 52a⟩≡ (112c) 52b ▷

```
static BSTR LispString2BSTR(LVAL val)
{
  char *s;
  int n;
  BSTR bs;

  if (! stringp(val)) xlbadtype(val);
  s = getstring(val);
  n = MultiByteToWideChar(CP_ACP, 0, s, -1, NULL, 0);
  if (n == 0) xlfail("conversion to BSTR failed");
  bs = SysAllocStringLen(NULL, n);
  if (bs == NULL) xlfail("BSTR allocation failed");
  MultiByteToWideChar(CP_ACP, 0, s, -1, bs, n);
  return bs;
}
```

52b ⟨*variant type conversion functions* 52a⟩+≡ (112c) ◁ 52a 54d ▷

```
LVAL BSTR2LispString(BSTR *bs)
{
  LVAL val;
  int n = WideCharToMultiByte(CP_ACP, 0, bs, -1, NULL, 0, NULL, NULL);
  if (n == 0) xlfail("BSTR conversion failed");
  val = newstring(n - 1);
  WideCharToMultiByte(CP_ACP, 0, bs, -1, getstring(val), n, NULL, NULL);
  return val;
}
```

52c ⟨*type conversion declarations* 41a⟩+≡ (112b) ◁ 41c

```
LVAL BSTR2LispString(BSTR *bs);
```

4.5.6 Structure to Variant Conversion

Conversion of Lisp structures to `VARIANT`s is handled by a callback.

52d ⟨*structure and object conversion* 52d⟩≡ (47d) 52f ▷

```
(wrap:c-callback "LispStruct2Variant" struct-to-variant
                 (:lval vartype (:cptr "VARIANTARG") bool) :void
                 :static nil)
```

52e ⟨*variant type conversion declarations* 51d⟩+≡ (112b) ◁ 51d 54b ▷

```
void LispStruct2Variant(LVAL val, VARTYPE type, VARIANTARG *pvar, BOOL ref_ok);
```

The callback handles `variant` structures and automation/com object and server structures. For variants, a copt is made if the variant argument is already the right type; otherwise `VariantChangeTypeEx` is used. Just to be safe, I'm using the `VariantCopyInd` function to force resolving of indirections–I don't think it is possible for there to be any, given where the variant structures come form, but I'm not sure.

52f ⟨*structure and object conversion* 52d⟩+≡ (47d) ◁ 52d 53a ▷

```
(defun struct-to-variant (val type pvar ref-ok)
  (etypecase val
    (variant
     (let ((ptr (variant-pointer val)))
       (if (or (= type VT_VARIANT) (= type (variant-type val)))
```

```
            (base-variant-copy-ind pvar ptr)
          (base-variant-change-type-ex pvar ptr (com-locale) 0 type))))
     (variant-ref
      (unless ref-ok (error "variant reference not allowed"))
      (let ((ref-val (variant-ref-variant val)))
        (unless (or (= type VT_VARIANT) (= type (variant-type ref-val)))
          (error "can't convert variant type in a reference"))
        (variant-ref-copy (variant-pointer ref-val) pvar)))
     (idispatch
      (com-pointer-to-variant val (idispatch-pointer val) type pvar t))
     (iunknown
      (com-pointer-to-variant val (iunknown-pointer val) type pvar nil))
     (auto-server
      (com-pointer-to-variant val (auto-server-pointer val) type pvar t))
     (generic-server
      (com-pointer-to-variant val (generic-server-pointer val) type pvar nil))))
```

**** document this stuff

53a ⟨*structure and object conversion* 52d⟩+≡ (47d) ◁52f 53b▷

```
(wrap:c-lines "
static void VariantRefCopy(VARIANTARG *prefvar, VARIANTARG *pvar)
{
  VariantClear(pvar);
  V_VT(pvar) = V_VT(prefvar) | VT_BYREF;
  V_I4REF(pvar) = &(V_I4(prefvar));
}")
(wrap:c-function variant-ref-copy "VariantRefCopy"
                 ((:cptr "VARIANTARG") (:cptr "VARIANTARG")) :void)
```

Interface pointers are all handled by essentially the same code, so it is broken out into a separate function. The value itself is passed to allow an error message to be constructed if necessary. The cast to an `IDispatch` poinbter is only needed for automation server objects. **** Maybe those should just be represented as `IDispatch` pointers as well.

53b ⟨*structure and object conversion* 52d⟩+≡ (47d) ◁53a 53c▷

```
(defun com-pointer-to-variant (val ptr type pvar dispatch)
  (cond
   ((and dispatch (or (= type VT_VARIANT) (= type VT_DISPATCH)))
    (lisp-to-variant-ex (cast-idispatch ptr) VT_DISPATCH pvar))
   ((= type VT_UNKNOWN)
    (lisp-to-variant-ex (cast-iunknown ptr) VT_UNKNOWN pvar))
   (t (error "can't convert ~a to ~a VARIANT"
             (type-of val)
             (variant-type-to-string type)))))
```

53c ⟨*structure and object conversion* 52d⟩+≡ (47d) ◁53b 53d▷

```
(wrap:c-pointer "IUnknown" (:cast cast-iunknown))
```

The `IDispatch` casting function is defined by

53d ⟨*structure and object conversion* 52d⟩+≡ (47d) ◁53c 53e▷

```
(wrap:c-pointer "IDispatch" (:cast cast-idispatch))
```

Even though I only use `VariantCopyInd`, I've also defined an interface for `VariantCopy` just in case i change my mind.

53e ⟨*structure and object conversion* 52d⟩+≡ (47d) ◁53d 54a▷
```
(wrap:std-com-function base-variant-copy "VariantCopy"
                       ((:cptr "VARIANTARG") (:cptr "VARIANTARG")))
(wrap:std-com-function base-variant-copy-ind "VariantCopyInd"
                       ((:cptr "VARIANTARG") (:cptr "VARIANTARG")))
```

4.5.7 Object to Variant Conversion

A callback hook is provided for converting a Lisp object to a `VARIANT`.

54a ⟨*structure and object conversion* 52d⟩+≡ (47d) ◁53e 54c▷
```
(wrap:c-callback "LispObject2Variant" object-to-variant
                 (:lval vartype (:cptr "VARIANTARG")) :void
                 :static nil)
```

54b ⟨*variant type conversion declarations* 51d⟩+≡ (112b) ◁52e 59a▷
```
void LispObject2Variant(LVAL val, VARTYPE type, VARIANTARG *pvar);
```

For the moment, the callback just signals an error. To allow Lisp objects to be put on the COM bus, just provide an appropriate definition for the `object-to-variant` function.

54c ⟨*structure and object conversion* 52d⟩+≡ (47d) ◁54a
```
(defun object-to-variant (object type pvar)
  (error "can't convert Lisp objects to VARIANTs"))
```

4.5.8 Safe Array to Lisp Array Conversion

This is needed because `SafeArrays` are stored in column-major order.

54d ⟨*variant type conversion functions* 52a⟩+≡ (112c) ◁52b 54e▷
```
static long calculate_index(long k, int rank, long *dims, long *idx)
{
  long nk, face;
  int i;

  for (i = 0, face = 1; i < rank; i++)
    face *= dims[i];

  for (i = 0; i < rank; i++) {
    face /= dims[i];
    idx[i] = k / face;
    k = k % face;
  }

  for (i = 0, nk = 0; i < rank; i++)
    nk = dims[rank - i - 1] * nk + idx[rank - i - 1];

  return nk;
}
```

**** incrase to 32 or 64? check limis in xlisp

54e ⟨*variant type conversion functions* 52a⟩+≡ (112c) ◁54d 56a▷
```
#define MAX_RANK 10
#define AS_PTR(t,p) ((t *) (p))

static LVAL SafeArray2Lisp(SAFEARRAY *psa, VARTYPE vt) {
```

```
LVAL val, dims, elem;
long lower, upper, size, i, n;
int rank = SafeArrayGetDim(psa);
long idims[MAX_RANK];
long idx[MAX_RANK];
void *sadata;
HRESULT hr;

if (rank > MAX_RANK) xlfail("array rank is too large to convert");

xlstkcheck(2);
xlsave(val);
xlsave(dims);
dims = rank == 1 ? NIL : newvector(rank);
for (size = 1, i = 1; i <= rank; i++) {
  SafeArrayGetLBound(psa, i, &lower);
  SafeArrayGetUBound(psa, i, &upper);
  n = upper - lower + 1;
  idims[i - 1] = n;
  size *= n;
  if (rank != 1) setelement(dims, i - 1, cvfixnum((FIXTYPE) n));
}

val = newvector(size);
hr = SafeArrayAccessData(psa, &sadata);
if (SUCCEEDED(hr)) {
  BEGIN_PROTECT {
    for (i = 0; i < size; i++) {
      long j = calculate_index(i, rank, idims, idx);
      switch (vt) {
      case VT_UI1: elem = cvfixnum(AS_PTR(unsigned char, sadata)[j]); break;
      case VT_I2: elem = cvfixnum(AS_PTR(short, sadata)[j]); break;
      case VT_I4: elem = cvfixnum(AS_PTR(long, sadata)[j]); break;
      case VT_R4: elem = cvflonum(AS_PTR(float, sadata)[j]); break;
      case VT_R8: elem = cvflonum(AS_PTR(double, sadata)[j]); break;
      case VT_CY:
        {
          CY cy = AS_PTR(CY, sadata)[j];
          elem = CY2Lisp(cy.s.Hi, cy.s.Lo);
        }
        break;
      case VT_DATE: elem = cvflonum(AS_PTR(DATE, sadata)[j]); break;
      case VT_BSTR: elem = BSTR2LispString(AS_PTR(BSTR, sadata)[j]); break;
      case VT_DISPATCH:
        {
          IDispatch *pdisp = AS_PTR(IDispatch *, sadata)[j];
          if (pdisp != NULL) IDispatch_AddRef(pdisp);
          elem = IDispatch2Lisp(pdisp);
        }
        break;
      case VT_ERROR: elem = ulong2lisp(AS_PTR(SCODE, sadata)[j]); break;
      case VT_BOOL:
        elem = AS_PTR(VARIANT_BOOL, sadata)[j]?s_true : NIL;
```

```
          break;
        case VT_VARIANT:
          elem = Variant2Lisp(&AS_PTR(VARIANT, sadata)[j]);
          break;
        case VT_UNKNOWN:
          {
            IUnknown *pdisp = AS_PTR(IUnknown *, sadata)[j];
            if (pdisp != NULL) IUnknown_AddRef(pdisp);
            elem = IUnknown2Lisp(pdisp);
          }
          break;
        default: xlfail("can't convert array");
        }
        setelement(val, i, elem);
      }
    }
    BEGIN_CLEANUP {
      SafeArrayUnaccessData(psa);
    }
    END_PROTECT
  }
  else xlfail("failed to access array data");

  if (rank != 1) val = newdarray(dims, val);
  xlpopn(2);
  return val;
}
```

4.5.9 Lisp Array to Safe Array Conversion

56a ⟨*variant type conversion functions* 52a⟩+≡ (112c) ◁54e 56b▷

```
DECLARE_CPTR_TYPE(void)
DECLARE_CPTR_TYPE(IDispatch)
DECLARE_CPTR_TYPE(IUnknown)

static IDispatch *Lisp2IDispatch(LVAL val)
{
  if (! cptr_type_p(val, CPTR_TYPE(IDispatch)) || getcpaddr(val) == NULL)
    xlbadtype(val);
  return getcpaddr(val);
}

static IUnknown *Lisp2IUnknown(LVAL val)
{
  if (! cptr_type_p(val, CPTR_TYPE(IUnknown)) || getcpaddr(val) == NULL)
    xlbadtype(val);
  return getcpaddr(val);
}
```

56b ⟨*variant type conversion functions* 52a⟩+≡ (112c) ◁56a 58▷

```
static SAFEARRAY *LispArray2SafeArray(LVAL val, VARTYPE vt) {
  LVAL data, elem;
  long size, i, n, rank, idims[MAX_RANK], idx[MAX_RANK];
```

```
SAFEARRAY *psa;
SAFEARRAYBOUND sabounds[MAX_RANK];
void *sadata;
HRESULT hr;

switch (ntype(val)) {
case STRING:
case VECTOR:
case TVEC:
  data = val;
  rank = 1;
  size = gettvecsize(data);
  sabounds[0].lLbound = 0;
  sabounds[0].cElements = size;
  break;
case DARRAY:
  {
    LVAL dims = getdarraydim(val);
    data = getdarraydata(val);
    rank = getdarrayrank(val);
    for (size = 1, i = 0; i < rank; i++) {
      n = getfixnum(getelement(dims, i));
      idims[i] = n;
      size *= n;
      sabounds[i].lLbound = 0;
      sabounds[i].cElements = n;
    }
  }
  break;
default: xlbadtype(val);
}

if (rank > MAX_RANK) xlfail("array rank is too large to convert");
if (rank == 0) xlfail("can't convert zero-rank arrays");

psa = SafeArrayCreate(vt, rank, sabounds);
if (psa == NULL) xlfail("SafeArray creation failed");

hr = SafeArrayAccessData(psa, &sadata);
if (SUCCEEDED(hr)) {
  BEGIN_PROTECT {
    xlsave1(elem);
    for (i = 0; i < size; i++) {
      long j = calculate_index(i, rank, idims, idx);
      elem = gettvecelement(data, i);
      switch (vt) {
      case VT_UI1:
        AS_PTR(unsigned char, sadata)[j] = lisp2ulong(elem);
        break;
      case VT_I2: AS_PTR(short, sadata)[j] = lisp2long(elem); break;
      case VT_I4: AS_PTR(long, sadata)[j] = lisp2long(elem); break;
      case VT_R4: AS_PTR(float, sadata)[j] = makefloat(elem); break;
      case VT_R8: AS_PTR(double, sadata)[j] = makefloat(elem); break;
```

```
        case VT_CY: Lisp2CY(elem, &AS_PTR(double, sadata)[j]); break;
        case VT_DATE: AS_PTR(DATE, sadata)[j] = makefloat(elem); break;
        case VT_BSTR: AS_PTR(BSTR, sadata)[j] = LispString2BSTR(elem); break;
        case VT_DISPATCH:
          {
            IDispatch *pdisp = Lisp2IDispatch(elem);
            if (pdisp != NULL) IDispatch_AddRef(pdisp);
            AS_PTR(IDispatch *, sadata)[j] = pdisp;
          }
          break;
        case VT_ERROR: AS_PTR(SCODE, sadata)[j] = lisp2ulong(elem); break;
        case VT_BOOL:
          AS_PTR(VARIANT_BOOL, sadata)[j] = null(elem)?0 : -1;
          break;
        case VT_VARIANT:
          Lisp2Variant(elem, VT_VARIANT, &AS_PTR(VARIANT, sadata)[j], FALSE);
          break;
        case VT_UNKNOWN:
          {
            IUnknown *punk = Lisp2IUnknown(elem);
            if (punk != NULL) IUnknown_AddRef(punk);
            AS_PTR(IUnknown *, sadata)[j] = punk;
          }
          break;
        default: xlfail("can only create VARIANT arrays for now");
        }
      }
      xlpop();
    }
    BEGIN_CLEANUP {
      SafeArrayUnaccessData(psa);
      if (UNWINDING) SafeArrayDestroy(psa);
    }
    END_PROTECT
  }
  else {
    SafeArrayDestroy(psa);
    xlfail("failed to access array data");
  }
  return psa;
}
```

4.5.10 Lisp to Variant Conversion

This uses VT_VARIANT as wild card since it is not a legal type for a VARIANT.

58 ⟨*variant type conversion functions* 52a⟩+≡ (112c) ◁56b 59c▷

```
void Lisp2Variant(LVAL val, VARTYPE type, VARIANTARG *pvar, BOOL ref_ok)
{
  if (structp(val))
    LispStruct2Variant(val, type, pvar, ref_ok);
  else if (objectp(val))
    LispObject2Variant(val, type, pvar);
```

```
    else if (type == VT_VARIANT) {
      switch (ntype(val)) {
      case SYMBOL:
        if (val == s_true || val == NIL) Lisp2VariantEx(val, VT_BOOL, pvar);
        else xlbadtype(val);
        break;
      case FIXNUM: Lisp2VariantEx(val, VT_I4, pvar); break;
      case BIGNUM:
      case FLONUM: Lisp2VariantEx(val, VT_R8, pvar); break;
      case STRING: Lisp2VariantEx(val, VT_BSTR, pvar); break;
      case VECTOR:
      case TVEC:
      case DARRAY: Lisp2VariantEx(val, VT_ARRAY | VT_VARIANT, pvar); break;
      default: xlbadtype(val);
      }
    }
    else Lisp2VariantEx(val, type, pvar);
  }
```

59a ⟨*variant type conversion declarations* 51d⟩+≡ (112b) ◁54b 60a▷

```
  void Lisp2Variant(LVAL val, VARTYPE type, VARIANTARG *pvar, BOOL ref_ok);
```

59b ⟨*Lisp to variant conversion* 59b⟩≡ (47d) 60b▷

```
  (wrap:c-function lisp-to-variant "Lisp2Variant"
                   (:lval vartype (:cptr "VARIANTARG") bool)
                   :void)
```

59c ⟨*variant type conversion functions* 52a⟩+≡ (112c) ◁58 61f▷

```
  void Lisp2VariantEx(LVAL val, VARTYPE type, VARIANTARG *pvar)
  {
    VariantInit(pvar);
    switch (type) {
    case VT_EMPTY: break;
    case VT_NULL: break;
    case VT_UI1: V_UI1(pvar) = lisp2ulong(val); break;
    case VT_I2: V_I2(pvar) = lisp2long(val); break;
    case VT_I4: V_I4(pvar) = lisp2long(val); break;
    case VT_R4: V_R4(pvar) = makefloat(val); break;
    case VT_R8: V_R8(pvar) = makefloat(val); break;
    case VT_CY: Lisp2CY(val, &V_CY(pvar)); break;
    case VT_DATE: V_DATE(pvar) = makefloat(val); break;
    case VT_BSTR: V_BSTR(pvar) = LispString2BSTR(val); break;
    case VT_DISPATCH:
      V_DISPATCH(pvar) = Lisp2IDispatch(val);
      if (V_DISPATCH(pvar) != NULL) IDispatch_AddRef(V_DISPATCH(pvar));
      break;
    case VT_ERROR: V_ERROR(pvar) = lisp2ulong(val); break;
    case VT_BOOL: V_BOOL(pvar) = null(val) ? 0 : -1; break;
    case VT_UNKNOWN:
      V_UNKNOWN(pvar) = Lisp2IUnknown(val);
      if (V_UNKNOWN(pvar) != NULL) IUnknown_AddRef(V_UNKNOWN(pvar));
      break;
    default:
      if (VT_ARRAY & type)
```

```
      V_ARRAY(pvar) = LispArray2SafeArray(val, type & ~VT_ARRAY);
    else xlfail("conversion not supported");
  }
  V_VT(pvar) = type;
}
```

60a ⟨*variant type conversion declarations* 51d⟩+≡ (112b) ◁59a 61e▷

```
void Lisp2VariantEx(LVAL val, VARTYPE type, VARIANTARG *pvar);
```

60b ⟨*Lisp to variant conversion* 59b⟩+≡ (47d) ◁59b 60c▷

```
(wrap:c-function lisp-to-variant-ex "Lisp2VariantEx"
                 (:lval vartype (:cptr "VARIANTARG"))
                 :void)
```

4.5.11 Variant References

60c ⟨*Lisp to variant conversion* 59b⟩+≡ (47d) ◁60b 60e▷

```
(defstruct (variant-ref (:constructor base-make-variant-ref (variant))
                        (:print-function print-variant-ref))
  variant)
```

60d ⟨*variant exports* 48d⟩+≡ (47d) ◁51c 60g▷

```
variant-ref
```

60e ⟨*Lisp to variant conversion* 59b⟩+≡ (47d) ◁60c 60f▷

```
(defun print-variant-ref (ref stream d)
  (let* ((var (variant-ref-variant ref))
         (ptr (when var (variant-pointer var)))
         (type (when ptr (base-variant-type ptr))))
    (format stream "#<~a~@[, type = ~a~]>"
            (type-of ref)
            (when type (variant-type-to-string type)))))
```

60f ⟨*Lisp to variant conversion* 59b⟩+≡ (47d) ◁60e 60h▷

```
(defun make-variant-ref (value type)
  (initialize-com)
  (let ((var (cond
              ((null value)
               (let ((ptr (new-variant)))
                 (set-variant-type ptr type)
                 (wrap-variant ptr)))
              ((variant-p value) value)
              (t (make-variant value type)))))
    (base-make-variant-ref var)))
```

60g ⟨*variant exports* 48d⟩+≡ (47d) ◁60d 60i▷

```
make-variant-ref
```

60h ⟨*Lisp to variant conversion* 59b⟩+≡ (47d) ◁60f 61a▷

```
(defun variant-ref-value (ref)
  (variant-value (variant-ref-variant ref)))
```

60i ⟨*variant exports* 48d⟩+≡ (47d) ◁60g 61c▷

```
variant-ref-value
```

61a ⟨*Lisp to variant conversion* 59b⟩+≡ (47d) ◁60h

```
(wrap:c-lines "
static void SetVariantType(VARIANTARG *pvar, VARTYPE type)
{
  V_VT(pvar) = type;
}")
(wrap:c-function set-variant-type "SetVariantType"
                 ((:cptr "VARIANTARG") vartype) :void)
```

***** need to check that reference counts work right

4.5.12 Variant to Lisp Conversion

**** is something like this needed for missing arguments too (`VT_ERROR` and `DISP_E_PARAMNOTFOUND`)?

61b ⟨*variant to Lisp conversion* 61b⟩≡ (47d) 61d▷

```
(defvar *null-variant-value* :null)
(defvar *empty-variant-value* :empty)
```

61c ⟨*variant exports* 48d⟩+≡ (47d) ◁60i 63b▷

```
*null-variant-value*
*empty-variant-value*
```

61d ⟨*variant to Lisp conversion* 61b⟩+≡ (47d) ◁61b 62b▷

```
(wrap:c-read-variable "GetNullVARIANT" *null-variant-value* :lval :static nil)
(wrap:c-read-variable "GetEmptyVARIANT" *empty-variant-value* :lval :static nil)
```

61e ⟨*variant type conversion declarations* 51d⟩+≡ (112b) ◁60a 62a▷

```
LVAL GetNullVARIANT(void);
LVAL GetEmptyVARIANT(void);
```

61f ⟨*variant type conversion functions* 52a⟩+≡ (112c) ◁59c

```
#define VAR_VAL(t,x) (V_ISBYREF(x) ? *V_##t##REF(x) : V_##t(x))
LVAL Variant2Lisp(VARIANT *pvar)
{
  LVAL val;
  if (V_ISARRAY(pvar)) {
    VARTYPE vt = V_VT(pvar) & ~ (VT_ARRAY|VT_BYREF);
    val = SafeArray2Lisp(VAR_VAL(ARRAY, pvar), vt);
  }
  else {
    switch (V_VT(pvar) & ~VT_BYREF) {
    case VT_EMPTY: val = GetEmptyVARIANT(); break;
    case VT_NULL: val = GetNullVARIANT(); break;
    case VT_UI1: val = cvfixnum(VAR_VAL(UI1, pvar)); break;
    case VT_I2: val = cvfixnum(VAR_VAL(I2, pvar)); break;
    case VT_I4: val = cvfixnum(VAR_VAL(I4, pvar)); break;
    case VT_R4: val = cvflonum(VAR_VAL(R4, pvar)); break;
    case VT_R8: val = cvflonum(VAR_VAL(R8, pvar)); break;
    case VT_CY: val = CY2Lisp(V_CY(pvar).s.Hi, V_CY(pvar).s.Lo); break;
    case VT_DATE: val = cvflonum(VAR_VAL(DATE, pvar)); break;
    case VT_BSTR: val = BSTR2LispString(V_BSTR(pvar)); break;
    case VT_DISPATCH:
      {
        IDispatch *pdisp = VAR_VAL(DISPATCH, pvar);
```

```
        if (pdisp != NULL) IDispatch_AddRef(pdisp);
        val = IDispatch2Lisp(pdisp);
      }
      break;
    case VT_ERROR: val = ulong2lisp(VAR_VAL(ERROR, pvar)); break;
    case VT_BOOL: val = VAR_VAL(BOOL, pvar) ? s_true : NIL; break;
    case VT_UNKNOWN:
      {
        IUnknown *pdisp = VAR_VAL(UNKNOWN, pvar);
        if (pdisp != NULL) IUnknown_AddRef(pdisp);
        val = IUnknown2Lisp(pdisp);
      }
      break;
    default: xlfail("unsupported variant type");
    }
  }
  return val;
}
```

62a ⟨*variant type conversion declarations* 51d⟩+≡ (112b) ◁61e

```
LVAL Variant2Lisp(VARIANT *pvar);
```

62b ⟨*variant to Lisp conversion* 61b⟩+≡ (47d) ◁61d

```
(wrap:c-function variant-to-lisp "Variant2Lisp" ((:cptr "VARIANTARG")) :lval)
```

4.5.13 Variant Collections

62c ⟨*dispatch constants* 42f⟩+≡ (25b) ◁51a

```
(wrap:c-constant DISPID_NEWENUM "DISPID_NEWENUM" :integer)
```

62d ⟨*variant collections* 62d⟩≡ (47d) 62e▷

```
(wrap:c-lines "
#define GetEnumVARIANT(a,b) IUnknown_QueryInterface(a,&IID_IEnumVARIANT,b)")
(wrap:std-com-function base-get-enum-variant "GetEnumVARIANT"
                       ((:cptr "IUnknown") (:value (:cptr "IEnumVARIANT"))))
(wrap:c-function release-enum-variant "IEnumVARIANT_Release"
                 ((:cptr "IEnumVARIANT")) :void)
```

**** just use query-interface here??

62e ⟨*variant collections* 62d⟩+≡ (47d) ◁62d 62f▷

```
(defun get-enum-variant (object)
  (let ((enum (invoke object DISPID_NEWENUM)))
    (base-get-enum-variant (iunknown-pointer enum))))
```

62f ⟨*variant collections* 62d⟩+≡ (47d) ◁62e 63a▷

```
(wrap:c-lines "
#define GetNextVARIANT(a,b) IEnumVARIANT_Next(a,1,b,NULL)")
(wrap:c-function enum-variant-next "GetNextVARIANT"
                 ((:cptr "IEnumVARIANT") (:cptr "VARIANTARG")) hresult)
```

**** lift unwind-protect out of loop?

63a ⟨*variant collections* 62d⟩+≡ (47d) ◁62f

```
(defmacro do-collection ((var object &optional value) &body body)
  (let ((colsym (gensym))
        (varsym (gensym)))
    `(let ((,colsym (get-enum-variant ,object)))
       (unwind-protect
           (let ((,varsym (new-variant)))
             (variant-init ,varsym)
             (loop
              (unwind-protect
                  (progn
                    (when (/= (enum-variant-next ,colsym ,varsym) S_OK)
                      (let ((,var nil)) (return ,value)))
                    (let ((,var (variant-to-lisp ,varsym))) ,@body))
                (variant-clear ,varsym))))
         (release-enum-variant ,colsym)))))
```

63b ⟨*variant exports* 48d⟩+≡ (47d) ◁61c

```
do-collection
```

63c ⟨*test* 47c⟩+≡ ◁47c 92b▷

```
(let* ((file "\\users\\luke\\working\\win32\\win32com\\cars.xls")
       (workbook (get-object file))
       (sheets (property workbook :worksheets))
       (val nil))
  (do-collection (s sheets (nreverse val))
    (push (property s :name) val)))
```

5 Type Libraries and Type Information

63d ⟨*type libraries and type information* 63d⟩≡ (112a)
⟨*type libraries* 63e⟩
⟨*type information* 69f⟩
⟨*finding coclasses* 80⟩

5.1 Type Libraries

63e ⟨*type libraries* 63e⟩≡ (63d)
(export '(⟨*type library exports* 64b⟩))
⟨*loading type libraries* 64a⟩
⟨*searching type libraries by name* 64c⟩
⟨*type library documentation* 65b⟩
⟨*type library attributes* 66a⟩
⟨*type library entries* 68a⟩
⟨*higher level type library functions* 69b⟩

5.1.1 Loading Type Libraries

64a ⟨*loading type libraries* 64a⟩≡ (63e)
```
(wrap:std-com-function base-load-type-lib "LoadTypeLib"
                       ((:cptr "WCHAR") (:value (:cptr "ITypeLib"))))

(defun load-type-lib (path)
  (initialize-com)
  (system:without-interrupts
   (wrap-itypelib (base-load-type-lib (string-to-wide-string path)))))
```

64b ⟨*type library exports* 64b⟩≡ (63e) 65a ▷
```
load-type-lib
```

5.1.2 Searching by Name

64c ⟨*searching type libraries by name* 64c⟩≡ (63e) 64d ▷
```
;;**** use pointer wrappers
;;**** allow for unsigned's in wrappers?
(wrap:c-pointer (:unsigned "int")
                (:make make-c-uint)
                (:get get-c-uint nil :integer)
                (:set set-c-uint nil :integer))

(wrap:c-pointer (:cptr "ITypeInfo")
                (:make make-type-info-array)
                (:get get-type-info-element nil (:cptr "ITypeInfo")))

(wrap:std-com-function base-itypelib-find-name "ITypeLib_FindName"
                       ((:cptr "ITypeLib") (:cptr "WCHAR") :integer
                        (:cptr (:cptr "ITypeInfo")) (:cptr "DISPID")
                        (:cptr (:unsigned "int"))))
```

64d ⟨*searching type libraries by name* 64c⟩+≡ (63e) ◁ 64c
```
(defun itypelib-find-name (lib name &optional (count 1 count-supplied-p))
  (let ((plib (itypelib-pointer lib))
        (wname (string-to-wide-string (string name)))
        (ptis (make-type-info-array count))
        (pids (make-dispid count))
        (pcount (make-c-uint 1)))
    (set-c-uint pcount count)
    (base-itypelib-find-name plib wname 0 ptis pids pcount)
    (let ((nfound (get-c-uint pcount)))
      (cond
       ((= nfound 0) (values nil nil))
       (count-supplied-p
        (let ((infos nil)
              (memids nil))
          (dotimes (i nfound (values (nreverse infos) (nreverse memids)))
            (push (wrap-itypeinfo (get-type-info-element ptis i)) infos)
            (push (get-dispid pids i) memids))))
       (t (values (wrap-itypeinfo (get-type-info-element ptis))
                  (get-dispid pids)))))))
```

65a ⟨*type library exports* 64b⟩+≡ (63e) ◁ 64b 65c ▷
```
itypelib-find-name
```

5.1.3 Documentation

65b ⟨*type library documentation* 65b⟩≡ (63e) 65d ▷
```
(wrap:std-com-function base-itypelib-documentation "ITypeLib_GetDocumentation"
                       ((:cptr "ITypeLib") :integer
                        (:cptr "BSTR" t) (:cptr "BSTR" t)
                        (:cptr "long" t) (:cptr "BSTR" t)))

(defun type-lib-documentation (lib &optional index all)
  (let ((i (if index index -1))
        (ptr (itypelib-pointer lib)))
    (get-documentation #'base-itypelib-documentation ptr i all)))

(defun itypelib-name (lib)
  (let ((ptr (itypelib-pointer lib)))
    (get-documentation #'base-itypelib-documentation ptr -1 nil)))
```

65c ⟨*type library exports* 64b⟩+≡ (63e) ◁ 65a 66e ▷
```
type-lib-documentation itypelib-name
```

65d ⟨*type library documentation* 65b⟩+≡ (63e) ◁ 65b 65e ▷
```
(wrap:c-pointer "BSTR"
                (:make make-bstr-array)
                (:get get-bstr nil (:cptr "WCHAR")))

(wrap:c-pointer "long" (:make make-long-array) (:get get-long nil :integer))

(wrap:c-function free-bstr "SysFreeString" ((:cptr "WCHAR")) :void)

(wrap:c-constant MEMBERID_NIL "MEMBERID_NIL" :integer)
```

65e ⟨*type library documentation* 65b⟩+≡ (63e) ◁ 65d
```
;;;**** assume that on failure BSTR's are null
(defun get-documentation (fun ptr index all)
  (system:without-interrupts
   (let* ((bsa-name (make-bstr-array))
          (bsa-doc (when all (make-bstr-array)))
          (la-help (when all (make-long-array)))
          (bsa-help (when all (make-bstr-array))))
     (funcall fun ptr index bsa-name bsa-doc la-help bsa-help)
     (let ((bs-name (get-bstr bsa-name))
           (bs-doc (when all (get-bstr bsa-doc)))
           (bs-help (when all (get-bstr bsa-help))))
       (unwind-protect
           (let ((name (if bs-name (wide-string-to-string bs-name) nil)))
             (if all
                 (values name
                         (when bs-doc (wide-string-to-string bs-doc))
                         (get-long la-help)
                         (when bs-help (wide-string-to-string bs-help)))
```

```
              name))
        (when bs-name (free-bstr bs-name))
        (when bs-doc (free-bstr bs-doc))
        (when bs-help (free-bstr bs-help)))))))
```

5.1.4 Attributes

66a ⟨*type library attributes* 66a⟩≡ (63e) 66b▷

```
(wrap:std-com-function get-tlibattr "ITypeLib_GetLibAttr"
                       ((:cptr "ITypeLib") (:value (:cptr "TLIBATTR"))))

(wrap:c-function release-tlibattr "ITypeLib_ReleaseTLibAttr"
                 ((:cptr "ITypeLib") (:cptr "TLIBATTR")) :void)
```

66b ⟨*type library attributes* 66a⟩+≡ (63e) ◁66a 66c▷

```
(wrap:c-pointer "TLIBATTR"
                (:get tlibattr-lcid "lcid" :integer)
                (:get tlibattr-syskind "syskind" :integer)
                (:get tlibattr-major "wMajorVerNum" :integer)
                (:get tlibattr-minor "wMinorVerNum" :integer)
                (:get tlibattr-flags "wLibFlags" :integer))

(wrap:c-lines "
static void GetTLibAttrGUID(TLIBATTR *pta, GUID *pid)
{
  *pid = pta->guid;
}")
(wrap:c-function tlibattr-get-guid "GetTLibAttrGUID"
                 ((:cptr "TLIBATTR") (:cptr "GUID")) :void)
```

66c ⟨*type library attributes* 66a⟩+≡ (63e) ◁66b 66d▷

```
(defmacro with-tlibattr ((asym lib) &body body)
  (let ((psym (gensym)))
    `(system:without-interrupts
      (let* ((,psym (itypelib-pointer ,lib))
             (,asym (get-tlibattr ,psym)))
        (unwind-protect
            (progn ,@body)
          (release-tlibattr ,psym ,asym))))))
```

66d ⟨*type library attributes* 66a⟩+≡ (63e) ◁66c 66f▷

```
(defun itypelib-guid (lib)
  (with-tlibattr (attr lib)
    (let ((guid (make-guid-data)))
      (tlibattr-get-guid attr guid)
      (make-guid guid))))
```

66e ⟨*type library exports* 64b⟩+≡ (63e) ◁65c 67a▷

```
itypelib-guid
```

66f ⟨*type library attributes* 66a⟩+≡ (63e) ◁66d 67b▷

```
(defun itypelib-lcid (lib)
  (with-tlibattr (attr lib) (tlibattr-lcid attr)))
```

67a ⟨*type library exports* 64b⟩+≡ (63e) ◁66e 67c▷
```
itypelib-lcid
```

67b ⟨*type library attributes* 66a⟩+≡ (63e) ◁66f 67e▷
```
(defun itypelib-syskind (lib)
  (with-tlibattr (attr lib)
    (let ((syskind (tlibattr-syskind attr)))
      (cond
       ((= syskind SYS_WIN16) :win16)
       ((= syskind SYS_WIN32) :win32)
       ((= syskind SYS_MAC) :mac)
       (t syskind)))))
```

67c ⟨*type library exports* 64b⟩+≡ (63e) ◁67a 67f▷
```
itypelib-syskind
```

67d ⟨*type library constants* 67d⟩≡ (25b) 67k▷
```
(wrap:c-constant SYS_WIN16 "SYS_WIN16" :integer)
(wrap:c-constant SYS_WIN32 "SYS_WIN32" :integer)
(wrap:c-constant SYS_MAC "SYS_MAC" :integer)
```

67e ⟨*type library attributes* 66a⟩+≡ (63e) ◁67b 67g▷
```
(defun itypelib-major-version (lib)
  (with-tlibattr (attr lib) (tlibattr-major attr)))
```

67f ⟨*type library exports* 64b⟩+≡ (63e) ◁67c 67h▷
```
itypelib-major-version
```

67g ⟨*type library attributes* 66a⟩+≡ (63e) ◁67e 67i▷
```
(defun itypelib-minor-version (lib)
  (with-tlibattr (attr lib) (tlibattr-minor attr)))
```

67h ⟨*type library exports* 64b⟩+≡ (63e) ◁67f 67j▷
```
itypelib-minor-version
```

67i ⟨*type library attributes* 66a⟩+≡ (63e) ◁67g
```
(defun itypelib-flags (lib)
  (with-tlibattr (attr lib)
    (let ((flags (tlibattr-flags attr))
          (lflags nil))
      (when (/= 0 (logand flags LIBFLAG_FRESTRICTED))
        (push :restricted lflags))
      (when (/= 0 (logand flags LIBFLAG_FHIDDEN)) (push :hidden lflags))
      (when (/= 0 (logand flags LIBFLAG_FCONTROL)) (push :control lflags))
      lflags)))
```

67j ⟨*type library exports* 64b⟩+≡ (63e) ◁67h 68b▷
```
itypelib-flags
```

67k ⟨*type library constants* 67d⟩+≡ (25b) ◁67d
```
(wrap:c-constant LIBFLAG_FRESTRICTED "LIBFLAG_FRESTRICTED" :integer)
(wrap:c-constant LIBFLAG_FCONTROL "LIBFLAG_FCONTROL" :integer)
(wrap:c-constant LIBFLAG_FHIDDEN "LIBFLAG_FHIDDEN" :integer)
```

5.1.5 Type Information Entries

68a ⟨*type library entries* 68a⟩≡ (63e) 68c ▷
```
(wrap:std-com-function base-itypelib-type-info "ITypeLib_GetTypeInfo"
                 ((:cptr "ITypeLib") :integer (:value (:cptr "ITypeInfo"))))

(defun itypelib-type-info (lib index)
  (wrap-itypeinfo (base-itypelib-type-info (itypelib-pointer lib) index)))
```

68b ⟨*type library exports* 64b⟩+≡ (63e) ◁ 67j 68d ▷
```
itypelib-type-info
```

68c ⟨*type library entries* 68a⟩+≡ (63e) ◁ 68a 68e ▷
```
(wrap:c-function base-itypelib-type-info-count "ITypeLib_GetTypeInfoCount"
                 ((:cptr "ITypeLib")) :unsigned)

(defun itypelib-type-info-count (lib)
  (base-itypelib-type-info-count (itypelib-pointer lib)))
```

68d ⟨*type library exports* 64b⟩+≡ (63e) ◁ 68b 68f ▷
```
itypelib-type-info-count
```

68e ⟨*type library entries* 68a⟩+≡ (63e) ◁ 68c 68h ▷
```
(wrap:std-com-function base-itypelib-type-info-of-guid
                       "ITypeLib_GetTypeInfoOfGuid"
                       ((:cptr "ITypeLib") (:cptr "GUID")
                        (:value (:cptr "ITypeInfo"))))

(defun itypelib-type-info-of-guid (lib guid)
  (let ((plib (itypelib-pointer lib)))
    (wrap-itypeinfo (base-itypelib-type-info-of-guid plib (guid-data guid)))))
```

68f ⟨*type library exports* 64b⟩+≡ (63e) ◁ 68d 68i ▷
```
itypelib-type-info-of-guid
```

68g ⟨*C type declarations* 31e⟩+≡ (25b) ◁ 50d 77b ▷
```
(wrap:declare-c-type typekind "TYPEKIND" :unsigned)
```

68h ⟨*type library entries* 68a⟩+≡ (63e) ◁ 68e 68j ▷
```
(wrap:std-com-function base-itypelib-type-info-type "ITypeLib_GetTypeInfoType"
                       ((:cptr "ITypeLib") :integer (:value typekind)))

(defun itypelib-type-info-type (lib index)
  (let ((plib (itypelib-pointer lib)))
    (type-kind-to-keyword (base-itypelib-type-info-type plib index))))
```

68i ⟨*type library exports* 64b⟩+≡ (63e) ◁ 68f 69a ▷
```
itypelib-type-info-type
```

68j ⟨*type library entries* 68a⟩+≡ (63e) ◁ 68h
```
(wrap:std-com-function base-itypelib-is-name "ITypeLib_IsName"
                 ((:cptr "ITypeLib") (:cptr "WCHAR") :integer (:value bool)))

(defun itypelib-is-name (lib name)
  (let ((plib (itypelib-pointer lib))
        (wname (string-to-wide-string name)))
    (base-itypelib-is-name plib wname 0)))
```

69a ⟨*type library exports* 64b⟩+≡ (63e) ◁68i 69c▷
```
itypelib-is-name
```

5.1.6 Higher Level Functions

69b ⟨*higher level type library functions* 69b⟩≡ (63e) 69d▷
```
(defun itypelib-infos-of-kind (lib kind)
  (let ((val nil))
    (dotimes (i (itypelib-type-info-count lib) (nreverse val))
      (when (eq kind (itypelib-type-info-type lib i))
        (push (itypelib-type-info lib i) val)))))
```

69c ⟨*type library exports* 64b⟩+≡ (63e) ◁69a 69e▷
```
itypelib-infos-of-kind
```

**** could avoid wrapping the info **** allow for multiple uses of name, check for constant

69d ⟨*higher level type library functions* 69b⟩+≡ (63e) ◁69b
```
(defun itypelib-find-constant (lib name)
  (multiple-value-bind (info memid) (itypelib-find-name lib name)
    (when info
      (dotimes (i (itypeinfo-variable-count info))
        (with-vardesc (vd info i)
          (when (= (vardesc-memid vd) memid)
            (return (vardesc-value vd))))))))
```

69e ⟨*type library exports* 64b⟩+≡ (63e) ◁69c
```
itypelib-find-constant
```

5.2 Type Information

69f ⟨*type information* 69f⟩≡ (63d)

(export '(⟨*type information exports* 70a⟩))
⟨*containing type library* 69g⟩
⟨*type information documentation* 70b⟩
⟨*type information attributes* 70d⟩
⟨*type information function descriptions* 73j⟩
⟨*type information member IDs from names* 75c⟩
⟨*type information implementation type flags* 76a⟩
⟨*type information names* 76c⟩
⟨*type information referenced types* 76e⟩
⟨*type information variable descriptions* 77e⟩
⟨*higher level type information functions* 79c⟩

5.2.1 Containing Type Library

69g ⟨*containing type library* 69g⟩≡ (69f)
```
(wrap:std-com-function base-itypeinfo-type-lib
                       "ITypeInfo_GetContainingTypeLib"
                       ((:cptr "ITypeInfo") (:value (:cptr "ITypeLib"))
                        (:cptr (:unsigned "int"))))

(defun itypeinfo-type-lib (info)
   (let* ((pti (itypeinfo-pointer info))
```

```
          (pidx (make-c-uint))
          (lib (system:without-interrupts
                (wrap-itypelib (base-itypeinfo-type-lib pti pidx)))))
     (values lib (get-c-uint pidx))))
```

70a ⟨*type information exports* 70a⟩≡ (69f) 70c ▷

```
itypeinfo-type-lib
```

5.2.2 Documentation

70b ⟨*type information documentation* 70b⟩≡ (69f)

```
(wrap:std-com-function base-itypeinfo-documentation
                       "ITypeInfo_GetDocumentation"
                       ((:cptr "ITypeInfo") :integer
                        (:cptr "BSTR" t) (:cptr "BSTR" t)
                        (:cptr "long" t) (:cptr "BSTR" t)))

(defun itypeinfo-documentation (info &optional index all)
  (let ((i (if index index MEMBERID_NIL))
        (ptr (itypeinfo-pointer info)))
    (get-documentation #'base-itypeinfo-documentation ptr i all)))

(defun itypeinfo-name (info)
  (let ((ptr (itypeinfo-pointer info)))
    (get-documentation #'base-itypeinfo-documentation ptr MEMBERID_NIL nil)))
```

70c ⟨*type information exports* 70a⟩+≡ (69f) ◁ 70a 71c ▷

```
itypeinfo-documentation itypeinfo-name
```

5.2.3 Type Information Attributes

70d ⟨*type information attributes* 70d⟩≡ (69f) 70e ▷

```
(wrap:std-com-function get-typeattr "ITypeInfo_GetTypeAttr"
                       ((:cptr "ITypeInfo") (:value (:cptr "TYPEATTR"))))

(wrap:c-function release-typeattr "ITypeInfo_ReleaseTypeAttr"
                 ((:cptr "ITypeInfo") (:cptr "TYPEATTR")) :void)
```

70e ⟨*type information attributes* 70d⟩+≡ (69f) ◁ 70d 71a ▷

```
(wrap:c-pointer "TYPEATTR"
                (:get typeattr-lcid "lcid" :integer)
                (:get typeattr-kind "typekind" :integer)
                (:get typeattr-cfuncs "cFuncs" :integer)
                (:get typeattr-cvars "cVars" :integer)
                (:get typeattr-cimpltypes "cImplTypes" :integer)
                (:get typeattr-flags "wTypeFlags" :integer)
                (:get typeattr-major "wMajorVerNum" :integer)
                (:get typeattr-minor "wMinorVerNum" :integer)
                (:get typeattr-alias "tdescAlias.hreftype" :integer)
                (:get typeattr-idlinfo "idldescType.wIDLFlags" :integer))

(wrap:c-lines "
static void GetTypeAttrGUID(TYPEATTR *pta, GUID *pid)
```

```
{
  *pid = pta->guid;
}")
(wrap:c-function typeattr-get-guid "GetTypeAttrGUID"
                 ((:cptr "TYPEATTR") (:cptr "GUID")) :void)
```

71a ⟨*type information attributes* 70d⟩+≡ (69f) ◁70e 71b▷

```
(defmacro with-typeattr ((asym info) &body body)
  (let ((psym (gensym)))
    `(system:without-interrupts
      (let* ((,psym (itypeinfo-pointer ,info))
             (,asym (get-typeattr ,psym)))
        (unwind-protect
            (progn ,@body)
          (release-typeattr ,psym ,asym))))))
```

71b ⟨*type information attributes* 70d⟩+≡ (69f) ◁71a 71d▷

```
(defun itypeinfo-guid (info)
  (with-typeattr (attr info)
    (let ((guid (make-guid-data)))
      (typeattr-get-guid attr guid)
      (make-guid guid))))
```

71c ⟨*type information exports* 70a⟩+≡ (69f) ◁70c 71e▷

```
itypeinfo-guid
```

71d ⟨*type information attributes* 70d⟩+≡ (69f) ◁71b 71g▷

```
(defun itypeinfo-lcid (info)
  (with-typeattr (attr info) (typeattr-lcid attr)))
```

71e ⟨*type information exports* 70a⟩+≡ (69f) ◁71c 72b▷

```
itypeinfo-lcid
```

71f ⟨*type information constants* 71f⟩≡ (25b) 72i▷

```
(wrap:c-constant TKIND_ENUM "TKIND_ENUM" :integer)
(wrap:c-constant TKIND_RECORD "TKIND_RECORD" :integer)
(wrap:c-constant TKIND_MODULE "TKIND_MODULE" :integer)
(wrap:c-constant TKIND_INTERFACE "TKIND_INTERFACE" :integer)
(wrap:c-constant TKIND_DISPATCH "TKIND_DISPATCH" :integer)
(wrap:c-constant TKIND_COCLASS "TKIND_COCLASS" :integer)
(wrap:c-constant TKIND_ALIAS "TKIND_ALIAS" :integer)
(wrap:c-constant TKIND_UNION "TKIND_UNION" :integer)
```

71g ⟨*type information attributes* 70d⟩+≡ (69f) ◁71d 72a▷

```
(defun type-kind-to-keyword (kind)
  (cond
   ((= kind TKIND_ENUM) :enum)
   ((= kind TKIND_RECORD) :record)
   ((= kind TKIND_MODULE) :module)
   ((= kind TKIND_INTERFACE) :interface)
   ((= kind TKIND_DISPATCH) :dispatch)
   ((= kind TKIND_COCLASS) :coclass)
   ((= kind TKIND_ALIAS) :alias)
   ((= kind TKIND_UNION) :union)
   (t :unknown)))
```

72a ⟨*type information attributes* 70d⟩+≡ (69f) ◁71g 72c▷
```
(defun itypeinfo-kind (info)
  (with-typeattr (attr info) (type-kind-to-keyword (typeattr-kind attr))))
```

72b ⟨*type information exports* 70a⟩+≡ (69f) ◁71e 72d▷
```
itypeinfo-kind
```

72c ⟨*type information attributes* 70d⟩+≡ (69f) ◁72a 72e▷
```
(defun itypeinfo-function-count (info)
  (with-typeattr (attr info) (typeattr-cfuncs attr)))
```

72d ⟨*type information exports* 70a⟩+≡ (69f) ◁72b 72f▷
```
itypeinfo-function-count
```

72e ⟨*type information attributes* 70d⟩+≡ (69f) ◁72c 72g▷
```
(defun itypeinfo-variable-count (info)
  (with-typeattr (attr info) (typeattr-cvars attr)))
```

72f ⟨*type information exports* 70a⟩+≡ (69f) ◁72d 72h▷
```
itypeinfo-variable-count
```

72g ⟨*type information attributes* 70d⟩+≡ (69f) ◁72e 72j▷
```
(defun itypeinfo-implementation-count (info)
  (with-typeattr (attr info) (typeattr-cimpltypes attr)))
```

72h ⟨*type information exports* 70a⟩+≡ (69f) ◁72f 73a▷
```
itypeinfo-implementation-count
```

72i ⟨*type information constants* 71f⟩+≡ (25b) ◁71f 74c▷
```
(wrap:c-constant TYPEFLAG_FAPPOBJECT "TYPEFLAG_FAPPOBJECT" :integer)
(wrap:c-constant TYPEFLAG_FCANCREATE "TYPEFLAG_FCANCREATE" :integer)
(wrap:c-constant TYPEFLAG_FLICENSED "TYPEFLAG_FLICENSED" :integer)
(wrap:c-constant TYPEFLAG_FHIDDEN "TYPEFLAG_FHIDDEN" :integer)
(wrap:c-constant TYPEFLAG_FCONTROL "TYPEFLAG_FCONTROL" :integer)
(wrap:c-constant TYPEFLAG_FDUAL "TYPEFLAG_FDUAL" :integer)
(wrap:c-constant TYPEFLAG_FNONEXTENSIBLE "TYPEFLAG_FNONEXTENSIBLE" :integer)
(wrap:c-constant TYPEFLAG_FOLEAUTOMATION "TYPEFLAG_FOLEAUTOMATION" :integer)
(wrap:c-constant TYPEFLAG_FRESTRICTED "TYPEFLAG_FRESTRICTED" :integer)
(wrap:c-constant TYPEFLAG_FAGGREGATABLE "TYPEFLAG_FAGGREGATABLE" :integer)
(wrap:c-constant TYPEFLAG_FREPLACEABLE "TYPEFLAG_FREPLACEABLE" :integer)
(wrap:c-constant TYPEFLAG_FDISPATCHABLE "TYPEFLAG_FDISPATCHABLE" :integer)
```

72j ⟨*type information attributes* 70d⟩+≡ (69f) ◁72g 73b▷
```
(defun itypeinfo-flags (info)
  (with-typeattr (attr info)
    (let ((flags (typeattr-flags attr))
          (lflags nil))
      (when (/= 0 (logand flags TYPEFLAG_FAPPOBJECT)) (push :appobject lflags))
      (when (/= 0 (logand flags TYPEFLAG_FCANCREATE)) (push :cancreate lflags))
      (when (/= 0 (logand flags TYPEFLAG_FLICENSED)) (push :licensed lflags))
      (when (/= 0 (logand flags TYPEFLAG_FHIDDEN)) (push :hidden lflags))
      (when (/= 0 (logand flags TYPEFLAG_FCONTROL)) (push :control lflags))
      (when (/= 0 (logand flags TYPEFLAG_FDUAL)) (push :dual lflags))
      (when (/= 0 (logand flags TYPEFLAG_FNONEXTENSIBLE))
        (push :nonextensible lflags))
      (when (/= 0 (logand flags TYPEFLAG_FOLEAUTOMATION))
```

```
        (push :oleautomation lflags))
      (when (/= 0 (logand flags TYPEFLAG_FRESTRICTED))
        (push :restricted lflags))
      (when (/= 0 (logand flags TYPEFLAG_FAGGREGATABLE))
        (push :aggregatable lflags))
      (when (/= 0 (logand flags TYPEFLAG_FREPLACEABLE))
        (push :replaceable lflags))
      (when (/= 0 (logand flags TYPEFLAG_FDISPATCHABLE))
        (push :dispatchable lflags))
      lflags)))
```

73a ⟨*type information exports* 70a⟩+≡ (69f) ◁72h 73c▷

```
itypeinfo-flags
```

73b ⟨*type information attributes* 70d⟩+≡ (69f) ◁72j 73d▷

```
(defun itypeinfo-major (info)
  (with-typeattr (attr info) (typeattr-major attr)))
```

73c ⟨*type information exports* 70a⟩+≡ (69f) ◁73a 73e▷

```
itypeinfo-major
```

73d ⟨*type information attributes* 70d⟩+≡ (69f) ◁73b 73f▷

```
(defun itypeinfo-minor (info)
  (with-typeattr (attr info) (typeattr-minor attr)))
```

73e ⟨*type information exports* 70a⟩+≡ (69f) ◁73c 73g▷

```
itypeinfo-minor
```

73f ⟨*type information attributes* 70d⟩+≡ (69f) ◁73d 73h▷

```
(defun itypeinfo-alias (info)
  (with-typeattr (attr info) (typeattr-alias attr)))
```

73g ⟨*type information exports* 70a⟩+≡ (69f) ◁73e 73i▷

```
itypeinfo-alias
```

73h ⟨*type information attributes* 70d⟩+≡ (69f) ◁73f

```
;;**** decode this??
(defun itypeinfo-idlinfo (info)
  (with-typeattr (attr info) (typeattr-idlinfo attr)))
```

73i ⟨*type information exports* 70a⟩+≡ (69f) ◁73g 75b▷

```
itypeinfo-idlinfo
```

5.2.4 Function Descriptions

73j ⟨*type information function descriptions* 73j⟩≡ (69f) 74a▷

```
(wrap:std-com-function get-funcdesc "ITypeInfo_GetFuncDesc"
                       ((:cptr "ITypeInfo") :integer
                        (:value (:cptr "FUNCDESC"))))

(wrap:c-function release-funcdesc "ITypeInfo_ReleaseFuncDesc"
                 ((:cptr "ITypeInfo") (:cptr "FUNCDESC")) :void)
```

74a ⟨*type information function descriptions* 73j⟩+≡ (69f) ◁73j 74b▷
```
(defmacro with-funcdesc ((fdsym info index) &body body)
  (let ((psym (gensym)))
    `(system:without-interrupts
      (let* ((,psym (itypeinfo-pointer ,info))
             (,fdsym (get-funcdesc ,psym ,index)))
        (unwind-protect
            (progn ,@body)
          (release-funcdesc ,psym ,fdsym))))))
```

**** decode vt's into keyword or symbol???

74b ⟨*type information function descriptions* 73j⟩+≡ (69f) ◁74a 74d▷
```
(wrap:c-pointer "FUNCDESC"
                (:get funcdesc-memid "memid" :integer)
                (:get funcdesc-params "lprgelemdescParam" (:cptr "ELEMDESC"))
                (:get base-funcdesc-funckind "funckind" :integer)
                (:get base-funcdesc-invkind "invkind" :integer)
                (:get funcdesc-cparams "cParams" :integer)
                (:get funcdesc-cparamsopt "cParamsOpt" :integer)
                (:get funcdesc-flags "wFuncFlags" :integer)
                (:get funcdesc-valtype "elemdescFunc.tdesc.vt" :integer))

(wrap:c-pointer "ELEMDESC" (:get elemdesc-valtype "tdesc.vt" :integer))
```

74c ⟨*type information constants* 71f⟩+≡ (25b) ◁72i 74e▷
```
(wrap:c-constant FUNC_PUREVIRTUAL "FUNC_PUREVIRTUAL" :integer)
(wrap:c-constant FUNC_VIRTUAL "FUNC_VIRTUAL" :integer)
(wrap:c-constant FUNC_NONVIRTUAL "FUNC_NONVIRTUAL" :integer)
(wrap:c-constant FUNC_STATIC "FUNC_STATIC" :integer)
(wrap:c-constant FUNC_DISPATCH "FUNC_DISPATCH" :integer)
```

74d ⟨*type information function descriptions* 73j⟩+≡ (69f) ◁74b 74f▷
```
(defun funcdesc-funckind (fd)
  (let ((funckind (base-funcdesc-funckind fd)))
    (cond
     ((= funckind FUNC_PUREVIRTUAL) :purevirtual)
     ((= funckind FUNC_VIRTUAL) :virtual)
     ((= funckind FUNC_NONVIRTUAL) :nonvirtual)
     ((= funckind FUNC_STATIC) :static)
     ((= funckind FUNC_DISPATCH) :dispatch)
     (t :unknown))))
```

74e ⟨*type information constants* 71f⟩+≡ (25b) ◁74c 75e▷
```
(wrap:c-constant INVOKE_FUNC "INVOKE_FUNC" :integer)
(wrap:c-constant INVOKE_PROPERTYGET "INVOKE_PROPERTYGET" :integer)
(wrap:c-constant INVOKE_PROPERTYPUT "INVOKE_PROPERTYPUT" :integer)
(wrap:c-constant INVOKE_PROPERTYPUTREF "INVOKE_PROPERTYPUTREF" :integer)
```

74f ⟨*type information function descriptions* 73j⟩+≡ (69f) ◁74d 75a▷
```
(defun funcdesc-invkind (fd)
  (let ((invkind (base-funcdesc-invkind fd)))
    (cond
     ((= invkind INVOKE_FUNC) :method)
     ((= invkind INVOKE_PROPERTYGET) :get)
```

```
      ((= invkind INVOKE_PROPERTYPUT) :put)
      ((= invkind INVOKE_PROPERTYPUTREF) :putref)
      (t :unknown))))
```

75a ⟨*type information function descriptions* 73j⟩+≡ (69f) ◁74f

```
  (defun itypeinfo-funcdesc (info i)
    (with-funcdesc (fd info i)
      (let ((memid (funcdesc-memid fd))
            (cparams (funcdesc-cparams fd)))
        (list memid
              (let ((ed (funcdesc-params fd))
                    (val nil))
                (dotimes (i cparams (nreverse val))
                  (push (elemdesc-valtype ed i) val)))
              (funcdesc-funckind fd)
              (funcdesc-invkind fd)
              cparams
              (funcdesc-cparamsopt fd)
              (funcdesc-valtype fd)
              (funcdesc-flags fd)
              (itypeinfo-names info memid (+ cparams 1))))))
```

75b ⟨*type information exports* 70a⟩+≡ (69f) ◁73i 75d▷

```
  itypeinfo-funcdesc
```

5.2.5 Member IDs From Names

75c ⟨*type information member IDs from names* 75c⟩≡ (69f)

```
  (wrap:std-com-function base-itypeinfo-ids-of-names "ITypeInfo_GetIDsOfNames"
                         ((:cptr "ITypeInfo") (:cptr (:cptr "WCHAR"))
                          :integer (:cptr "DISPID")))

  ;;**** share code with idispatch case?
  (defun itypeinfo-ids-of-names (info name &optional keys)
    (if name
        (let* ((names (cons name keys))
               (n (length names))
               (wnames (names-to-wide-string-array names))
               (dispids (make-dispid n))
               (pinfo (itypeinfo-pointer info)))
          (base-itypeinfo-ids-of-names pinfo wnames n dispids)
          (let ((val (dispids-to-list dispids n)))
            (values (first val) (rest val))))
      (if keys
          (error "can't have named arguments with default mehtod")
        DISPID_VALUE)))
```

75d ⟨*type information exports* 70a⟩+≡ (69f) ◁75b 76b▷

```
  itypeinfo-ids-of-name
```

5.2.6 Implementation Type Flags

75e ⟨*type information constants* 71f⟩+≡ (25b) ◁74e 78b▷

```
(wrap:c-constant IMPLTYPEFLAG_FDEFAULT "IMPLTYPEFLAG_FDEFAULT" :integer)
(wrap:c-constant IMPLTYPEFLAG_FSOURCE "IMPLTYPEFLAG_FSOURCE" :integer)
(wrap:c-constant IMPLTYPEFLAG_FRESTRICTED "IMPLTYPEFLAG_FRESTRICTED" :integer)
(wrap:c-constant IMPLTYPEFLAG_FDEFAULTVTABLE "IMPLTYPEFLAG_FDEFAULTVTABLE"
                 :integer)
```

76a ⟨*type information implementation type flags* 76a⟩≡ (69f)

```
;;***** value type needs checking
(wrap:std-com-function base-itypeinfo-implflags "ITypeInfo_GetImplTypeFlags"
                       ((:cptr "ITypeInfo") :integer (:value :integer)))

(defun itypeinfo-implementation-flags (info index)
  (let ((flags (base-itypeinfo-implflags (itypeinfo-pointer info) index))
        (lflags nil))
    (when (/= 0 (logand flags IMPLTYPEFLAG_FDEFAULT)) (push :default lflags))
    (when (/= 0 (logand flags IMPLTYPEFLAG_FSOURCE)) (push :source lflags))
    (when (/= 0 (logand flags IMPLTYPEFLAG_FRESTRICTED))
      (push :restricted lflags))
    (when (/= 0 (logand flags IMPLTYPEFLAG_FDEFAULTVTABLE))
      (push :defaultvtable))
    lflags))
```

76b ⟨*type information exports* 70a⟩+≡ (69f) ◁75d 76d▷

```
itypeinfo-implementation-flags
```

*****GetMops?

5.2.7 Getting Names

**** Do main name as first value, rest as list in second value? **** make count into result value?

76c ⟨*type information names* 76c⟩≡ (69f)

```
(wrap:std-com-function base-itypeinfo-get-names "ITypeInfo_GetNames"
                       ((:cptr "ITypeInfo") :integer
                        (:cptr "BSTR") :integer (:cptr "long")))

(defun itypeinfo-names (info memid count)
  (let ((pcount (make-long-array))
        (wnames (make-bstr-array count))
        (pinfo (itypeinfo-pointer info)))
    (base-itypeinfo-get-names pinfo memid wnames count pcount)
    (let ((nres (get-long pcount))
          (val nil))
      (unwind-protect
          (dotimes (i nres (nreverse val))
            (push (wide-string-to-string (get-bstr wnames i)) val))
        (dotimes (i nres) (free-bstr (get-bstr wnames i)))))))
```

76d ⟨*type information exports* 70a⟩+≡ (69f) ◁76b 77a▷

```
itypeinfo-names
```

5.2.8 Referenced Types

76e ⟨*type information referenced types* 76e⟩≡ (69f) 77c▷

```
(wrap:std-com-function base-itypeinfo-ref-type-info "ITypeInfo_GetRefTypeInfo"
```

```
                         ((:cptr "ITypeInfo") :integer
                          (:value (:cptr "ITypeInfo"))))

  (defun itypeinfo-ref-type-info (info ref)
    (system:without-interrupts
     (wrap-itypeinfo
      (base-itypeinfo-ref-type-info (itypeinfo-pointer info) ref))))
```

77a ⟨*type information exports* 70a⟩+≡ (69f) ◁76d 77d▷

```
  itypeinfo-ref-type-info
```

77b ⟨*C type declarations* 31e⟩+≡ (25b) ◁68g

```
  (wrap:declare-c-type hreftype "HREFTYPE" :unsigned)
```

77c ⟨*type information referenced types* 76e⟩+≡ (69f) ◁76e

```
  (wrap:std-com-function base-itypeinfo-ref-type-of-impl-type
                         "ITypeInfo_GetRefTypeOfImplType"
                         ((:cptr "ITypeInfo") :integer (:value hreftype)))

  (defun itypeinfo-ref-type-of-impl-type (info index)
    (base-itypeinfo-ref-type-of-impl-type (itypeinfo-pointer info) index))
```

77d ⟨*type information exports* 70a⟩+≡ (69f) ◁77a 79b▷

```
  itypeinfo-ref-type-of-impl-type
```

5.2.9 Variable Descriptions

77e ⟨*type information variable descriptions* 77e⟩≡ (69f) 77f▷

```
  (wrap:std-com-function get-vardesc "ITypeInfo_GetVarDesc"
                         ((:cptr "ITypeInfo") :integer
                          (:value (:cptr "VARDESC"))))

  (wrap:c-function release-vardesc "ITypeInfo_ReleaseVarDesc"
                   ((:cptr "ITypeInfo") (:cptr "VARDESC")) :void)
```

77f ⟨*type information variable descriptions* 77e⟩+≡ (69f) ◁77e 77g▷

```
  (defmacro with-vardesc ((vdsym info index) &body body)
    (let ((psym (gensym)))
      `(system:without-interrupts
        (let* ((,psym (itypeinfo-pointer ,info))
               (,vdsym (get-vardesc ,psym ,index)))
          (unwind-protect
              (progn ,@body)
            (release-vardesc ,psym ,vdsym))))))
```

77g ⟨*type information variable descriptions* 77e⟩+≡ (69f) ◁77f 78a▷

```
  (wrap:c-pointer "VARDESC"
                  (:get vardesc-memid "memid" :integer)
                  (:get base-vardesc-varkind "varkind" :integer)
                  (:get base-vardesc-flags "wVarFlags" :integer)
                  (:get vardesc-type "elemdescVar.tdesc.vt" :integer))
```

78a ⟨*type information variable descriptions* 77e⟩+≡ (69f) ◁77g 78c▷
```
(wrap:c-lines "
static LVAL GetVarDescValue(VARDESC *pvd)
{
  if (pvd->varkind != VAR_CONST) xlfail(\"variable is not a constant\");
  return Variant2Lisp(pvd->lpvarValue);
}")
(wrap:c-function vardesc-value "GetVarDescValue" ((:cptr "VARDESC")) :lval)
```

78b ⟨*type information constants* 71f⟩+≡ (25b) ◁75e 78d▷
```
(wrap:c-constant VAR_PERINSTANCE "VAR_PERINSTANCE" :integer)
(wrap:c-constant VAR_STATIC "VAR_STATIC" :integer)
(wrap:c-constant VAR_CONST "VAR_CONST" :integer)
(wrap:c-constant VAR_DISPATCH "VAR_DISPATCH" :integer)
```

78c ⟨*type information variable descriptions* 77e⟩+≡ (69f) ◁78a 78e▷
```
(defun vardesc-varkind (vd)
  (let ((varkind (base-vardesc-varkind vd)))
    (cond
     ((= varkind VAR_PERINSTANCE) :perinstance)
     ((= varkind VAR_STATIC) :static)
     ((= varkind VAR_CONST) :const)
     ((= varkind VAR_DISPATCH) :dispatch)
     (t :unknown))))
```

78d ⟨*type information constants* 71f⟩+≡ (25b) ◁78b
```
(wrap:c-constant VARFLAG_FREADONLY "VARFLAG_FREADONLY" :integer)
(wrap:c-constant VARFLAG_FSOURCE "VARFLAG_FSOURCE" :integer)
(wrap:c-constant VARFLAG_FBINDABLE "VARFLAG_FBINDABLE" :integer)
(wrap:c-constant VARFLAG_FREQUESTEDIT "VARFLAG_FREQUESTEDIT" :integer)
(wrap:c-constant VARFLAG_FDISPLAYBIND "VARFLAG_FDISPLAYBIND" :integer)
(wrap:c-constant VARFLAG_FDEFAULTBIND "VARFLAG_FDEFAULTBIND" :integer)
(wrap:c-constant VARFLAG_FHIDDEN "VARFLAG_FHIDDEN" :integer)
(wrap:c-constant VARFLAG_FRESTRICTED "VARFLAG_FRESTRICTED" :integer)
(wrap:c-constant VARFLAG_FDEFAULTCOLLELEM "VARFLAG_FDEFAULTCOLLELEM" :integer)
(wrap:c-constant VARFLAG_FUIDEFAULT "VARFLAG_FUIDEFAULT" :integer)
(wrap:c-constant VARFLAG_FNONBROWSABLE "VARFLAG_FNONBROWSABLE" :integer)
(wrap:c-constant VARFLAG_FREPLACEABLE "VARFLAG_FREPLACEABLE" :integer)
(wrap:c-constant VARFLAG_FIMMEDIATEBIND "VARFLAG_FIMMEDIATEBIND" :integer)
```

78e ⟨*type information variable descriptions* 77e⟩+≡ (69f) ◁78c 79a▷
```
(defun vardesc-flags (vd)
  (let ((flags (base-vardesc-flags vd))
        (lflags nil))
    (when (/= 0 (logand flags VARFLAG_FREADONLY)) (push :readonly lflags))
    (when (/= 0 (logand flags VARFLAG_FSOURCE)) (push :source lflags))
    (when (/= 0 (logand flags VARFLAG_FBINDABLE)) (push :bindable lflags))
    (when (/= 0 (logand flags VARFLAG_FREQUESTEDIT))
      (push :requestedit lflags))
    (when (/= 0 (logand flags VARFLAG_FDISPLAYBIND))
      (push :displaybind lflags))
    (when (/= 0 (logand flags VARFLAG_FDEFAULTBIND))
      (push :defaultbind lflags))
    (when (/= 0 (logand flags VARFLAG_FHIDDEN)) (push :hidden lflags))
```

```
      (when (/= 0 (logand flags VARFLAG_FRESTRICTED)) (push :restricted lflags))
      (when (/= 0 (logand flags VARFLAG_FDEFAULTCOLLELEM))
        (push :defaultcollelem lflags))
      (when (/= 0 (logand flags VARFLAG_FUIDEFAULT)) (push :uidefault lflags))
      (when (/= 0 (logand flags VARFLAG_FNONBROWSABLE))
        (push :nonbrowsable lflags))
      (when (/= 0 (logand flags VARFLAG_FREPLACEABLE))
        (push :replaceable lflags))
      (when (/= 0 (logand flags VARFLAG_FIMMEDIATEBIND))
        (push :immediatebind lflags))
      lflags))
```

79a ⟨*type information variable descriptions* 77e⟩+≡ (69f) ◁ 78e

```
  (defun itypeinfo-vardesc (info i)
    (with-vardesc (vd info i)
      (let ((memid (vardesc-memid vd))
            (varkind (vardesc-varkind vd)))
        (list memid
              varkind
              (when (eq varkind :const) (vardesc-value vd))
              (vardesc-type vd)
              (vardesc-flags vd)
              (first (itypeinfo-names info memid 1))))))
```

79b ⟨*type information exports* 70a⟩+≡ (69f) ◁ 77d 79d ▷

```
  itypeinfo-vardesc
```

5.2.10 Higher Level Functions

79c ⟨*higher level type information functions* 79c⟩≡ (69f) 79e ▷

```
  (defun itypeinfo-function-descriptions (info)
    (let ((val nil))
      (dotimes (i (itypeinfo-function-count info) (nreverse val))
        (push (itypeinfo-funcdesc info i) val))))
```

79d ⟨*type information exports* 70a⟩+≡ (69f) ◁ 79b 79f ▷

```
  itypeinfo-function-descriptions
```

79e ⟨*higher level type information functions* 79c⟩+≡ (69f) ◁ 79c 79g ▷

```
  (defun itypeinfo-variable-descriptions (info)
    (let ((val nil))
      (dotimes (i (itypeinfo-variable-count info) (nreverse val))
        (push (itypeinfo-vardesc info i) val))))
```

79f ⟨*type information exports* 70a⟩+≡ (69f) ◁ 79d 79h ▷

```
  itypeinfo-variable-descriptions
```

79g ⟨*higher level type information functions* 79c⟩+≡ (69f) ◁ 79e

```
  (defun itypeinfo-implementations (info)
    (let ((val nil))
      (dotimes (i (itypeinfo-implementation-count info) (nreverse val))
        (let ((ref (itypeinfo-ref-type-of-impl-type info i)))
          (push (itypeinfo-ref-type-info info ref) val)))))
```

79h ⟨*type information exports* 70a⟩+≡ (69f) ◁ 79f

```
  itypeinfo-implementations
```

5.3 Finding CoClasses

**** raise error instead of returning `NULL`? The easy way ought to be to use `IProvideClassInfo`. But this doesn't seem to work for anything in Excel, so as an alternative we can search the type library for a coclass that contains the info. I'm doing matching on name; it would be better to match on GUID.

In any case, this may not really be needed to make events work. It may be enough to use the connection point enumeration to find what is supported. Typically there will be only one dispatch interface (I think). If so, it has got to be the one; if there are more, then maybe this exercise is needed to locate the default source.

I think this will work: Use the enumerator to find all dispatch connection interfaces. If there is more than one, either just pick the first or maybe then go in and look for the default one. Or require that the calles specify the name or GUID.

80 ⟨*finding coclasses* 80⟩≡ (63d)

```
(wrap:c-lines "
static ITypeInfo *GetCoClassInfo(IUnknown *punk)
{
  IProvideClassInfo *ppci;
  ITypeInfo *pti;
  HRESULT hr;

  hr = IUnknown_QueryInterface(punk, &IID_IProvideClassInfo, (void**)&ppci);
  if (SUCCEEDED(hr)) {
    hr = IProvideClassInfo_GetClassInfo(ppci, &pti);
    IUnknown_Release(ppci);
  }
  return SUCCEEDED(hr) ? pti : NULL;
}")
(wrap:c-function base-get-coclass-info "GetCoClassInfo" ((:cptr "IUnknown"))
                 (:cptr "ITypeInfo"))

(defun get-coclass-info (object)
  (let ((ccinfo-ptr (base-get-coclass-info (iunknown-pointer object))))
    (if ccinfo-ptr
        (wrap-itypeinfo ccinfo-ptr)
      (let* ((info (idispatch-type-info object))
             (guid (itypeinfo-guid info))
             (lib (itypeinfo-type-lib info)))
        (dotimes (i (itypelib-type-info-count lib))
          (when (eq (itypelib-type-info-type lib i) :coclass)
            (let ((ti (itypelib-type-info lib i)))
              (dotimes (i (itypeinfo-implementation-count ti))
                (let* ((ref (itypeinfo-ref-type-of-impl-type ti i))
                       (tii (itypeinfo-ref-type-info ti ref)))
                  (when (eq guid (itypeinfo-guid tii))
                    (return-from get-coclass-info ti))))))))))

(defun type-lib-coclass-infos (lib)
  (let* ((plib (itypelib-pointer lib))
         (n (base-type-lib-get-type-info-count plib))
         (val nil))
    (dotimes (i n (nreverse val))
      (when (= (base-type-lib-get-type-info-type plib i) TKIND_COCLASS)
        (system:without-interrupts
```

```
          (push (wrap-itypeinfo (base-type-lib-get-type-info plib i)) val))))))
```

6 Server Support

81a ⟨*COM server support* 81a⟩≡ (112a)
  ⟨*COM servers* 81b⟩
  ⟨*COM event handling* 92c⟩
  ⟨*COM class factories* 97c⟩
  ⟨*higher level server interface* 103c⟩

6.1 A Simple Server

This section presents the mechanism for exposing COM server objects.

81b ⟨*COM servers* 81b⟩≡ (81a)
```
  (export '(⟨server exports 81d⟩))
  ⟨server constant wrappers 84c⟩
  ⟨server function wrappers 82c⟩
  ⟨server lisp interface 81c⟩
```

6.1.1 Generic Servers

Generic servers provide a mechanism for exposing a COM object and managing its reference counting. This corresponds to the generic `IUnknown` interface.

At the Lisp level, a generic server is a structure containing a pointer slot for the internal representation.

81c ⟨*server lisp interface* 81c⟩≡ (81b) 82d ▷
```
  (defstruct (generic-server (:constructor new-generic-server (pointer))
                             (:print-function
                              (lambda (serv stream d)
                                (declare (ignore d))
                                (format stream "#<~a>" (type-of serv)))))
    pointer)
```

81d ⟨*server exports* 81d⟩≡ (81b) 85e ▷
```
  generic-server
```

The internal representation is a structure containing a Vtbl and some fields for managing reference counts.

81e ⟨*server support declarations* 81e⟩≡ (112b) 83b ▷
```
  typedef struct tagGenericServer GenericServer;

  typedef struct {
    ⟨IUnknown Vtbl entries 82a⟩
  } GenericServerVtbl;

  typedef struct tagGenericServer {
    GenericServerVtbl *lpVtbl;
    ⟨generic interface fields 82b⟩
  } GenericServer;
```

The Vtbl contains the methods required by the `IUnknown` interface.

82a ⟨`IUnknown` *Vtbl entries* 82a⟩≡ (81e 86g 98e)
```
HRESULT (STDMETHODCALLTYPE *QueryInterface)(GenericServer *This,
                                            REFIID riid, void **ppvObject);
ULONG (STDMETHODCALLTYPE *AddRef)(GenericServer *This);
ULONG (STDMETHODCALLTYPE *Release)(GenericServer *This);
```

The fields used by the internal representation consist of a reference count, a pointer to the containing Lisp object and a flag indicating whether the server is a class factory.

82b ⟨*generic interface fields* 82b⟩≡ (81e 87a 98f)
```
long count;
LVAL object;
BOOL isClassFac;
```

A function for casting a pointer to a generic server pointer is provided by

82c ⟨*server function wrappers* 82c⟩≡ (81b) 83c ▷
```
(wrap:c-pointer "GenericServer" (:cast cast-generic-server))
```

6.1.2 Reference Counting

The reference counting mechanism insures that the containing Lisp object, and through it the internal data, are protected from GC when the intrnal object is in use by a COM client.

The protected objects are stored as a list in the variable `*protected-com-servers*`.

82d ⟨*server lisp interface* 81c⟩+≡ (81b) ◁ 81c 84a ▷
```
(defvar win32-com::*protected-com-servers* nil)
```

At the C level, the symbol is looked up once and stored in a global variable.

82e ⟨*server support functions* 82e⟩≡ (112c) 82g ▷
```
static LVAL s_com_servers = NULL;
```

82f ⟨*look up* `*protected-com-servers*` *symbol* 82f⟩≡ (82g)
```
if (s_com_servers == NULL)
  s_com_servers = xlenter("WIN32-COM::*PROTECTED-COM-SERVERS*");
```

Protecting and unprotecting the server object is hanbled by

82g ⟨*server support functions* 82e⟩+≡ (112c) ◁ 82e 82h ▷
```
static void ProtectGenericServer(LVAL object)
{
  ⟨look up *protected-com-servers* symbol 82f⟩
  setvalue(s_com_servers, cons(object, getvalue(s_com_servers)));
}
static void UnprotectGenericServer(LVAL object)
{
  ⟨look up *protected-com-servers* symbol 82f⟩
  setvalue(s_com_servers, xldelete1(object, getvalue(s_com_servers)));
}
```

Access to the reference counts, and to the protected object list, are controled by a single critical section.

82h ⟨*server support functions* 82e⟩+≡ (112c) ◁ 82g 83a ▷
```
CRITICAL_SECTION server_protect_cs;
```

The reference counting methods are thus

83a ⟨*server support functions* 82e⟩+≡ (112c) ◁82h 87b▷
```
static STDMETHODIMP_(ULONG) Generic_AddRef(GenericServer *this)
{
  EnterCriticalSection(&server_protect_cs);
  if (this->count == 0) {
    ProtectGenericServer(this->object);
    if (! this->isClassFac) CoAddRefServerProcess();
  }
  this->count++;
  LeaveCriticalSection(&server_protect_cs);
#ifdef DEBUG
  {
    char *buf[256];
    sprintf(buf, "Reference count up to %d\n", this->count);
    stdputstr(buf);
  }
#endif /* DEBUG */
  return this->count;
}
static STDMETHODIMP_(ULONG) Generic_Release(GenericServer *this)
{
  EnterCriticalSection(&server_protect_cs);
  this->count--;
  if (this->count == 0) {
    UnprotectGenericServer(this->object);
    if (! this->isClassFac) MyReleaseServerProcess();
  }
  LeaveCriticalSection(&server_protect_cs);
#ifdef DEBUG
  {
    char *buf[256];
    sprintf(buf, "Reference count down to %d\n", this->count);
    stdputstr(buf);
  }
#endif /* DEBUG */
  return this->count;
}
int MyReleaseServerProcess()
{
  int count = CoReleaseServerProcess();
  if (count == 0) {
    XLGLOBAL extern Exiting;
    Exiting = TRUE;
    PostQuitMessage(0);
  }
  return count;
}
```
**** need to break out the debugging code

83b ⟨*server support declarations* 81e⟩+≡ (112b) ◁81e 86f▷
```
int MyReleaseServerProcess(void);
```

83c ⟨*server function wrappers* 82c⟩+≡ (81b) ◁82c 84b▷
```
(wrap:c-function add-ref-server-process "CoAddRefServerProcess" () :integer)
(wrap:c-function release-server-process "MyReleaseServerProcess" () :integer)
```

84a ⟨*server lisp interface* 81c⟩+≡ (81b) ◁82d 84f▷
```
(unless (fboundp 'base-main-frame-visible)
  (setf (symbol-function 'base-main-frame-visible) #'msw-main-frame-visible))
(defun msw-main-frame-visible (&optional (vis nil set))
  (if set
      (let ((current-vis (base-main-frame-visible)))
        (when (com-embedding)
          (cond
           ((and (not vis) current-vis) (release-server-process))
           ((and vis (not current-vis)) (add-ref-server-process))))
        (base-main-frame-visible vis))
    (base-main-frame-visible)))
```

6.1.3 Registering Active Objects

Any generic server can be registered with COM as an active object for a class.

Active objects are regiestered with `RegisterActiveObject` and revoked with `RevokeActiveObject`.

84b ⟨*server function wrappers* 82c⟩+≡ (81b) ◁83c 84d▷
```
(wrap:std-com-function base-register-active-object "RegisterActiveObject"
                       ((:cptr "IUnknown") (:cptr "GUID") dword
                        (:value ulong)))
(wrap:std-com-function base-revoke-active-object "RevokeActiveObject"
        (ulong (:cptr "void" t)))
```

Active objects can be registered with a strong or a weak lock.

84c ⟨*server constant wrappers* 84c⟩≡ (81b) 88d▷
```
(wrap:c-constant ACTIVEOBJECT_STRONG "ACTIVEOBJECT_STRONG" :unsigned)
(wrap:c-constant ACTIVEOBJECT_WEAK "ACTIVEOBJECT_WEAK" :unsigned)
```

The recommended approach is to use weak locks and then manage a strong lock with `CoLockObjectExternal`.

84d ⟨*server function wrappers* 82c⟩+≡ (81b) ◁84b 84e▷
```
(wrap:std-com-function co-lock-object-external "CoLockObjectExternal"
                       ((:cptr "IUnknown") bool bool))
```

Before shutdown, any connected clients should be disconnected with

84e ⟨*server function wrappers* 82c⟩+≡ (81b) ◁84d 88a▷
```
(wrap:std-com-function co-disconnect-object "CoDisconnectObject"
        ((:cptr "IUnknown") dword))
```

To manage a higher level interface a list is used to record information about registered active objects.

84f ⟨*server lisp interface* 81c⟩+≡ (81b) ◁84a 84g▷
```
(defparameter *active-objects* nil)
```

The entries consist of list structures.

84g ⟨*server lisp interface* 81c⟩+≡ (81b) ◁84f 85a▷
```
(defstruct (registration-entry
            (:constructor make-registration-entry (clsid server cookie))
            (:type list))
  clsid server cookie)
```

Entries are looked up with

85a ⟨*server lisp interface* 81c⟩+≡ (81b) ◁84g 85b▷
```
(defun find-active-object-entry (clsid)
  (find clsid *active-objects* :key #'registration-entry-clsid))
```
In addition, a list of all entries for a server is returned by

85b ⟨*server lisp interface* 81c⟩+≡ (81b) ◁85a 85c▷
```
(defun find-active-object-entries (server)
  (let ((val nil))
    (dolist (entry *active-objects* val)
      (when (eq server (registration-entry-server entry)) (push entry val)))))
```
Entries are created and removed by

85c ⟨*server lisp interface* 81c⟩+≡ (81b) ◁85b 85d▷
```
(defun enter-active-object (clsid server cookie)
  (when (find-active-object-entry clsid)
    (error "already have an active object for CLSID ~a" clsid))
  (push (make-registration-entry clsid server cookie) *active-objects*))

(defun remove-active-object-entry (entry)
  (setf *active-objects* (remove entry *active-objects*)))
```
For the moment, the public active object registration function revokes an existing registration, makes a weak registration of the new server, establishes a strong lock, and enters the server in the data base.

85d ⟨*server lisp interface* 81c⟩+≡ (81b) ◁85c 85f▷
```
(defun register-active-object (server cls-spec)
  (initialize-com)
  (let* ((clsid (find-clsid cls-spec))
         (entry (find-active-object-entry clsid)))
    (when entry
      (revoke-active-object (registration-entry-server entry) clsid))
    (let* ((punk (cast-iunknown (generic-server-pointer server)))
           (cookie (base-register-active-object punk (guid-data clsid)
                                                   ACTIVEOBJECT_WEAK)))
      (co-lock-object-external punk t t)
      (enter-active-object clsid server cookie)
      t)))
```
85e ⟨*server exports* 81d⟩+≡ (81b) ◁81d 86a▷
```
register-active-object
```
The public revocation function either revokes all entries for an object or only the one for the specified class. If the server is `nil` then the server for the specified class is revoked. The function first removes the server entry from the Lisp data base, removes the strong lock, then revokes the registration, and finally disconnects any client objects.

85f ⟨*server lisp interface* 81c⟩+≡ (81b) ◁85d 86b▷
```
(defun revoke-active-object (server &optional cls-spec)
  (flet ((revoke-entry (entry)
           (remove-active-object-entry entry)
           (let* ((server (registration-entry-server entry))
                  (punk (cast-iunknown (generic-server-pointer server)))
                  (cookie (registration-entry-cookie entry)))
             (co-lock-object-external punk nil t)
             (base-revoke-active-object cookie nil)
```

```
                (co-disconnect-object punk 0))))
      (if cls-spec
          (let ((entry (find-active-object-entry (find-clsid cls-spec))))
            (when (and entry
                       (or (null server)
                           (eq server (registration-entry-server entry))))
              (revoke-entry entry)))
        (dolist (entry (find-active-object-entries server))
          (revoke-entry entry)))))
```

86a ⟨*server exports* 81d⟩+≡ (81b) ◁85e 86e▷

```
revoke-active-object
```

All active objects can be revoked by calling `revoke-all-active-objects`. This is used when COM is uninitialized.

86b ⟨*server lisp interface* 81c⟩+≡ (81b) ◁85f 86c▷

```
(defun revoke-all-active-objects ()
  (dolist (entry *active-objects*)
    (let ((server (registration-entry-server entry))
          (clsid (registration-entry-clsid entry)))
      (revoke-active-object server clsid))))
```

All servers can be disconnected by calling `disconnect-all-servers`. This should only be called after active objects and class factories have been revoked.

86c ⟨*server lisp interface* 81c⟩+≡ (81b) ◁86b 86d▷

```
(defun disconnect-all-servers ()
  (dolist (server *protected-com-servers*)
     (let ((punk (cast-iunknown (generic-server-pointer server))))
       (co-disconnect-object punk 0))))
```

6.1.4 Automation Servers

Automation servers are represented at the Lisp level by a structure that inherits from th egeneric server structure and contains two additional slots, the callbacks to implement the `GetIDsOfNames` and `Invoke` `IDispatch` methods.

86d ⟨*server lisp interface* 81c⟩+≡ (81b) ◁86c 88c▷

```
(defstruct (auto-server
            (:include generic-server)
            (:constructor
             new-auto-server (pointer getids invoke error-source)))
  getids invoke error-source)
```

86e ⟨*server exports* 81d⟩+≡ (81b) ◁86a 92a▷

```
auto-server
```

The internal representation is a C structure.

86f ⟨*server support declarations* 81e⟩+≡ (112b) ◁83b 86g▷

```
typedef struct tagAutoServer AutoServer;
```

The Vtbl for the automation server contains the `IUnknown` methods of the generic server followed by the `IDispatch` methods.

86g ⟨*server support declarations* 81e⟩+≡ (112b) ◁86f 87a▷
```
typedef struct {
  ⟨IUnknown Vtbl entries 82a⟩
  HRESULT (STDMETHODCALLTYPE *GetTypeInfoCount)(AutoServer *This,
                                                UINT *pctinfo);
  HRESULT (STDMETHODCALLTYPE *GetTypeInfo)(AutoServer *This, UINT iTInfo,
                                           LCID lcid, ITypeInfo **ppTInfo);
  HRESULT (STDMETHODCALLTYPE *GetIDsOfNames)(AutoServer *This, REFIID riid,
                                             LPOLESTR *rgszNames, UINT cNames,
                                             LCID lcid, DISPID *rgDispId);
  HRESULT (STDMETHODCALLTYPE *Invoke)(AutoServer *This, DISPID dispIdMember,
                                      REFIID riid, LCID lcid, WORD wFlags,
                                      DISPPARAMS *pDispParams,
                                      VARIANT *pVarResult,
                                      EXCEPINFO *pExcepInfo, UINT *puArgErr);
} AutoServerVtbl;
```
The server structure is given by

87a ⟨*server support declarations* 81e⟩+≡ (112b) ◁86g 88b▷
```
struct tagAutoServer {
  AutoServerVtbl *lpVtbl;
  ⟨generic interface fields 82b⟩
};
```

6.1.5 Basic Automation Server Methods

The `QueryInterface` method signals an error unless the requested interface is `IDispatch` or `IUnknown`.

87b ⟨*server support functions* 82e⟩+≡ (112c) ◁83a 87c▷
```
static STDMETHODIMP
AutoServer_QueryInterface(GenericServer *this, REFIID riid, void **ppv)
{
  if (IsEqualIID(riid, &IID_IUnknown) || IsEqualIID(riid, &IID_IDispatch)) {
    IUnknown_AddRef((IUnknown *) this);
    *ppv = this;
    return S_OK;
  }
  else {
    *ppv = NULL;
    return E_NOINTERFACE;
  }
}
```
Type information is not supported yet. `GetTypeInfoCount` therefore returns a count of zero.

87c ⟨*server support functions* 82e⟩+≡ (112c) ◁87b 87d▷
```
static STDMETHODIMP AutoServer_GetTypeInfoCount(AutoServer *this, UINT *pn)
{
  *pn = 0;
  return NOERROR;
}
```
If `GetTypeInfo` is called anyway, an error is signaled. I'm not sure what the appropriate error is; the one used here is the one Brockschmidt uses [1, p. 680].

87d ⟨*server support functions* 82e⟩+≡ (112c) ◁87c 89a▷
```
static STDMETHODIMP AutoServer_GetTypeInfo(AutoServer *this, UINT i,
                                           LCID lcid, ITypeInfo **ppti)
{
  *ppti = NULL;
  return E_NOTIMPL;
}
```
It might be a good idea to put callback hooks in here now to allow type info to be handled at the Lisp level later.

6.1.6 Automation Server GetIDsOfNames Method

The `GetIDsOfNames` method is implemented with a Lisp callback. Interrupts are disallowed and non-local exits are trapped and result in a value of `E_UNEXPECTED`.

88a ⟨*server function wrappers* 82c⟩+≡ (81b) ◁84e 89b▷
```
(wrap:c-callback "ServerGetIDsOfNames" server-get-ids-of-names
                 (:lval (:cptr (:cptr "WCHAR")) :unsigned :unsigned
                        (:cptr "DISPID"))
                 hresult
                 :static nil
                 :interrupts-allowed nil
                 :trap-exits "E_UNEXPECTED")
```
88b ⟨*server support declarations* 81e⟩+≡ (112b) ◁87a 89c▷
```
HRESULT ServerGetIDsOfNames(LVAL server, WCHAR **wnames, unsigned long count,
                            unsigned long lcid, DISPID *dispids);
```
The callback function uses the function stored in the servers's `getids` slot. This function takes two arguments, the method name and a list of keyword names. It returns two values, the method index and a list of keyword indices. Unknown indices are represented by `nil`.

88c ⟨*server lisp interface* 81c⟩+≡ (81b) ◁86d 89e▷
```
(defun server-get-ids-of-names (server wnames count lcid dispids)
  (let ((val S_OK))
    (flet ((set-id (i v)
             (cond
              (v (set-dispid dispids v i))
              (t (set-dispid dispids DISPID_UNKNOWN i)
                 (setf val DISP_E_UNKNOWNNAME)))))
      (when (< 0 count)
        (let ((name (wide-string-to-string (get-cptr-wchar wnames 0)))
              (keys (let ((val nil))
                      (dotimes (i (- count 1) (nreverse val))
                        (let ((wstr (get-cptr-wchar wnames (+ i 1))))
                          (push (wide-string-to-string wstr) val)))))
              (fun (auto-server-getids server)))
          (multiple-value-bind (disp kdisps error)
                               (funcall fun server name keys lcid)
            (set-id 0 disp)
            (dotimes (i (- count 1)) (setid (+ i 1) (pop kdisps)))))))
    val))
```
The callback uses several new constants.

88d ⟨*server constant wrappers* 84c⟩+≡ (81b) ◁84c
```
(wrap:c-constant DISPID_UNKNOWN "DISPID_UNKNOWN" :unsigned)
(wrap:c-constant DISP_E_UNKNOWNNAME "DISP_E_UNKNOWNNAME" :unsigned)
(wrap:c-constant S_OK "S_OK" :unsigned)
```

The automation server `GetIDsOfNames` method just calls the callback wrapper with the server object and the method's arguments.

89a ⟨*server support functions* 82e⟩+≡ (112c) ◁87d 89d▷
```
static STDMETHODIMP AutoServer_GetIDsOfNames(AutoServer *this, REFIID riid,
                                             LPOLESTR *wNames, UINT cNames,
                                             LCID lcid, DISPID *pid)
{
  return ServerGetIDsOfNames(this->object, wNames, cNames, lcid, pid);
}
```

6.1.7 Automation Server Invoke Method

The `Invoke` method also uses a callback. This callback traps exits. It currently also disallows interrupts; this should probably be changed so runaway calculations can be killed.

89b ⟨*server function wrappers* 82c⟩+≡ (81b) ◁88a 90a▷
```
(wrap:c-callback "ServerInvoke" server-invoke
                 (:lval :unsigned :unsigned :unsigned
                        (:cptr "DISPPARAMS") (:cptr "VARIANTARG")
                        (:cptr "EXCEPINFO") (:cptr (:unsigned "int")))
                 hresult
                 :static nil
                 :interrupts-allowed nil
                 :trap-exits "E_UNEXPECTED")
```

89c ⟨*server support declarations* 81e⟩+≡ (112b) ◁88b 91c▷
```
HRESULT ServerInvoke(LVAL object, unsigned long dispid, unsigned long lcid,
                     unsigned long flags, DISPPARAMS *params, VARIANT *result,
                     EXCEPINFO *excep, unsigned int *argerr);
```

The `Invoke` method just calls the callback and returns its result.

89d ⟨*server support functions* 82e⟩+≡ (112c) ◁89a 91a▷
```
static STDMETHODIMP AutoServer_Invoke(AutoServer *this, DISPID id, REFIID riid,
                                      LCID lcid, WORD wFlags, DISPPARAMS *pdp,
                                      VARIANT *pv, EXCEPINFO *pei, UINT *pae)
{
  return ServerInvoke(this->object, id, lcid, wFlags, pdp, pv, pei, pae);
}
```

The Lisp callback extracts the argument information, calls the server's invoke function, and returns a result if a result is requested.

89e ⟨*server lisp interface* 81c⟩+≡ (81b) ◁88c 91f▷
```
(defun server-invoke (object dispid lcid flags params result excep argerr)
  (handler-case
     (let ((args nil)
           (keys nil)
           (fun (auto-server-invoke object)))
       (dotimes (i (dispparams-arg-count params))
         (push (dispparams-arg params i) args))
       (dotimes (i (dispparams-named-arg-count params))
```

```
          (push (dispparams-name-id params i) keys))
        (let* ((for-val (if result t nil))
               (val (funcall fun object dispid lcid flags args keys for-val)))
          (when result (lisp-to-variant val VT_VARIANT result nil)))
        S_OK)
      (error (c)
             (when excep
               (let ((source (auto-server-error-source object))
                     (desc (format nil "~a" c)))
                 (fill-excep excep source desc)))
             DISP_E_EXCEPTION)))
```

The argument information is obtained from the `DISPPARAMS` structure.

90a ⟨*server function wrappers* 82c⟩+≡ (81b) ◁89b 90b▷

```
(wrap:c-pointer "DISPPARAMS"
                (:get dispparams-arg-count "cArgs" :integer)
                (:get dispparams-named-arg-count "cNamedArgs" :integer)
                (:get dispparams-args "rgvarg" (:cptr "Variant"))
                (:get dispparams-name-parg-dispids "rgdispidNamedArgs"
                      (:cptr "DISPID")))
```

The arguments themselves are accessed using a separate function. This could in principle be handled by getting the `rgvarg` pointer, extracting separate variant pointers, and then using a Lisp-level version of `Variant2Lisp`. For now the present approach seems more efficient, but maybe I'll switch to the more generic approach once Lisp-level variant support is available.

90b ⟨*server function wrappers* 82c⟩+≡ (81b) ◁90a 90c▷

```
(wrap:c-lines "
static LVAL DispparamsArg(DISPPARAMS *dpars, int i)
{
  return Variant2Lisp(&(dpars->rgvarg[i]));
}")
(wrap:c-function dispparams-arg "DispparamsArg"
                 ((:cptr "DISPPARAMS") :integer) :lval)
```

Similarly, the keyword indices are accessed with a separate function. Here it would really make more sense to use a generic approach.

90c ⟨*server function wrappers* 82c⟩+≡ (81b) ◁90b 90d▷

```
(wrap:c-lines "
static DISPID DispparamsNameID(DISPPARAMS *dpars, int i)
{
  return dpars->rgdispidNamedArgs[i];
}")
(wrap:c-function dispparams-name-id "DispparamsNameID"
                 ((:cptr "DISPPARAMS") :integer) :integer)
```

Finally, we need to fill in the exception info in some useful way. I don't know what to use for the `scode`, so I just use `E_FAIL`. Presumably some standard Lisp errors could be mapped to some standard exceptions; maybe a look at the COM/CORBA books would help. For now, I just put the printed representation of the exception in the description field.

90d ⟨*server function wrappers* 82c⟩+≡ (81b) ◁90c 91d▷

```
(wrap:c-lines "
static void FillExcep(EXCEPINFO *excep, char *src, char *desc)
{
  int m = MultiByteToWideChar(CP_ACP, 0, src, -1, NULL, 0);
```

```
  int n = MultiByteToWideChar(CP_ACP, 0, desc, -1, NULL, 0);
  if ((excep->bstrSource = SysAllocStringLen(NULL, m)) != NULL)
    MultiByteToWideChar(CP_ACP, 0, src, -1, excep->bstrSource, m);
  if ((excep->bstrDescription = SysAllocStringLen(NULL, n)) != NULL)
    MultiByteToWideChar(CP_ACP, 0, desc, -1, excep->bstrDescription, n);
  excep->scode = E_FAIL;
}")
(wrap:c-function fill-excep "FillExcep" ((:cptr "EXCEPINFO") :string :string)
                 :void)
```

6.1.8 Automation Server Initialization

The Vtbl for automation servers is

91a ⟨*server support functions* 82e⟩+≡ (112c) ◁89d 91b▷

```
static AutoServerVtbl my_ServerVtbl = {
  AutoServer_QueryInterface,
  Generic_AddRef,
  Generic_Release,
  AutoServer_GetTypeInfoCount,
  AutoServer_GetTypeInfo,
  AutoServer_GetIDsOfNames,
  AutoServer_Invoke
};
```

The initialization function `InitializeAutoServer` installs the Vtbl and initializes the generic server fields.

91b ⟨*server support functions* 82e⟩+≡ (112c) ◁91a

```
void InitializeAutoServer(AutoServer *serv, LVAL object)
{
  serv->lpVtbl = &my_ServerVtbl;
  serv->count = 0;
  serv->object = object;
  serv->isClassFac = FALSE;
}
```

91c ⟨*server support declarations* 81e⟩+≡ (112b) ◁89c

```
void InitializeAutoServer(AutoServer *serv, LVAL object);
```

91d ⟨*server function wrappers* 82c⟩+≡ (81b) ◁90d 91e▷

```
(wrap:c-function init-auto-server "InitializeAutoServer"
                 ((:cptr "AutoServer") :lval) :void)
```

The constructor for internal automation server objects is defined by the pointer wrapper

91e ⟨*server function wrappers* 82c⟩+≡ (81b) ◁91d

```
(wrap:c-pointer "AutoServer" (:make base-make-my-server))
```

The constructor for a Lisp-level automation server creates and initializes an internal server and wraps it in the `auto-server` structure. The arguments to the constructor are the `GetIDsOfNames` and `Invoke` callback functions. The server object is placed on the pointer's protection list using `cptr-protect` to insure that the server object remains protected as long as the pointer object is reachable.

91f ⟨*server lisp interface* 81c⟩+≡ (81b) ◁89e 103b▷
```
(defun make-auto-server (getids invoke &optional (source "XlispStat"))
  (let* ((pointer (base-make-my-server))
         (server (new-auto-server pointer getids invoke source)))
    (system:cptr-protect pointer server)
    (init-auto-server pointer server)
    server))
```

92a ⟨*server exports* 81d⟩+≡ (81b) ◁86e
```
make-auto-server
```

92b ⟨*test* 47c⟩+≡ ◁63c 97a▷
```
(defun sample-getids (object name keys lcid)
  (format t "Requesting DISPID for ~s~%" name)
  0)
(defun sample-invoke (object dispid lcid flags args keys value)
  (if value
      (format t "Invoking method ~d for value~%" dispid)
    (format t "Invoking method ~d~%" dispid))
  (format t "Args: ~s~%Named args: ~s~%" args keys)
  0)
(setf s (make-auto-server #'sample-getids #'sample-invoke))
(register-active-object s "XlispStat.application")
(setf x (get-active-object "XlispStat.application"))
(property x :fred)
(revoke-active-object s)
```

6.2 Event Handling

92c ⟨*COM event handling* 92c⟩≡ (81a)
```
(export '(⟨event handling exports 94d⟩))
⟨connection point container functions 92d⟩
⟨connection point enumeration functions 93d⟩
⟨connection point functions 94f⟩
⟨connection point registration 95j⟩
```

6.2.1 Connection Point Containers

**** just use query-interface instead of raw version?? **** avoid lookup??

92d ⟨*connection point container functions* 92d⟩≡ (92c) 92e▷
```
(defun get-connection-point-container (ref)
  (let* ((intf (find-interface IID_IConnectionPointContainer))
         (iid-data (guid-data (interface-iid intf)))
         (caster (interface-caster intf)))
    (funcall caster (base-query-interface (iunknown-pointer ref) iid-data t))))
```

92e ⟨*connection point container functions* 92d⟩+≡ (92c) ◁92d 93a▷
```
(wrap:c-function release-connection-point-container
                 "IConnectionPointContainer_Release"
                 ((:cptr "IConnectionPointContainer")) :void)
```

93a ⟨*connection point container functions* 92d⟩+≡ (92c) ◁92e 93b▷
```
(wrap:std-com-function connection-point-container-enumerator
                       "IConnectionPointContainer_EnumConnectionPoints"
                       ((:cptr "IConnectionPointContainer")
                        (:value (:cptr "IEnumConnectionPoints"))))
```

93b ⟨*connection point container functions* 92d⟩+≡ (92c) ◁93a 93c▷
```
(wrap:std-com-function connection-point-container-find
                       "IConnectionPointContainer_FindConnectionPoint"
                 ((:cptr "IConnectionPointContainer") (:cptr "GUID")
                  (:value (:cptr "IConnectionPoint"))))
```

93c ⟨*connection point container functions* 92d⟩+≡ (92c) ◁93b
```
(defmacro with-connection-point-container ((cpc object) &body body)
  (let ((cpcsym (gensym)))
    `(let ((,cpcsym (get-connection-point-container ,object)))
       (unwind-protect
           (let ((,cpc ,cpcsym)) ,@body)
       (release-connection-point-container ,cpcsym)))))
```

6.2.2 Connection Point Enumeration

93d ⟨*connection point enumeration functions* 93d⟩≡ (92c) 93e▷
```
(wrap:c-function release-connection-point-enumerator
                 "IEnumConnectionPoints_Release"
                 ((:cptr "IEnumConnectionPoints")) :void)
```

93e ⟨*connection point enumeration functions* 93d⟩+≡ (92c) ◁93d 93f▷
```
(wrap:c-pointer (:cptr "IConnectionPoint")
                (:make make-connection-point-array)
                (:get get-connection-point-element nil
                      (:cptr "IConnectionPoint")))
```

93f ⟨*connection point enumeration functions* 93d⟩+≡ (92c) ◁93e 93g▷
```
;;**** use wrapptrs
(wrap:c-pointer (:unsigned "long")
                (:make make-c-ulong)
                (:get get-c-ulong nil :integer))
```

93g ⟨*connection point enumeration functions* 93d⟩+≡ (92c) ◁93f 93h▷
```
(wrap:c-function base-connection-point-enumerator-next
                 "IEnumConnectionPoints_Next"
                 ((:cptr "IEnumConnectionPoints") ulong
                  (:cptr (:cptr "IConnectionPoint"))
                  (:cptr (:unsigned "long")))
                 hresult)
```

93h ⟨*connection point enumeration functions* 93d⟩+≡ (92c) ◁93g 94a▷
```
;;**** enumerate seems to fail once exhausted???
(defun connection-point-enumerator-next (enum)
  (let ((pcpa (make-connection-point-array))
        (pul (make-c-ulong)))
    (let ((hr (base-connection-point-enumerator-next enum 1 pcpa pul)))
```

```
      (if (and (not (hresult-failed hr)) (= (get-c-ulong pul) 1))
          (get-connection-point-element pcpa)
        nil))))
```

94a ⟨*connection point enumeration functions* 93d⟩+≡ (92c) ◁93h 94b▷

```
;;Skip(ULONG n)
;;Reset()
;;Clone(IEnumConnectionPoints **new)
```

94b ⟨*connection point enumeration functions* 93d⟩+≡ (92c) ◁94a 94c▷

```
(defmacro do-connection-points ((cp object &optional val) &body body)
  (let ((cpcsym (gensym))
        (cpsym (gensym))
        (enumsym (gensym)))
    `(with-connection-point-container (,cpcsym ,object)
       (let ((,enumsym (connection-point-container-enumerator ,cpcsym)))
         (unwind-protect
             (loop
              (let ((,cpsym (connection-point-enumerator-next ,enumsym)))
                (when (null ,cpsym) (let ((,cp nil)) (return ,val)))
                (unwind-protect
                    (let ((,cp ,cpsym)) ,@body)
                  (release-connection-point ,cpsym))))
           (release-connection-point-enumerator ,enumsym))))))
```

94c ⟨*connection point enumeration functions* 93d⟩+≡ (92c) ◁94b 94e▷

```
(defun get-source-interfaces (object)
  (let ((val nil))
    (do-connection-points (cp object (nreverse val))
      (push (connection-point-interface cp) val))))
```

94d ⟨*event handling exports* 94d⟩≡ (92c) 95g▷

```
get-source-interfaces
```

94e ⟨*connection point enumeration functions* 93d⟩+≡ (92c) ◁94c

```
(defun get-source-infos (object)
  (let* ((lib (itypeinfo-type-lib (idispatch-type-info object)))
         (iids (get-source-interfaces object))
         (infos (mapcar #'(lambda (iid) (itypelib-type-info-of-guid lib iid))
                        iids)))
    (remove-if-not #'(lambda (x) (eq :dispatch (itypeinfo-kind x))) infos)))
```

6.2.3 Connection Points

94f ⟨*connection point functions* 94f⟩≡ (92c) 94g▷

```
(wrap:c-function release-connection-point "IConnectionPoint_Release"
                 ((:cptr "IConnectionPoint")) :void)
```

94g ⟨*connection point functions* 94f⟩+≡ (92c) ◁94f 95a▷

```
(wrap:std-com-function base-connection-point-interface
                       "IConnectionPoint_GetConnectionInterface"
                       ((:cptr "IConnectionPoint") (:cptr "GUID")))
```

95a ⟨*connection point functions* 94f⟩+≡ (92c) ◁94g 95b▷
```
(defun connection-point-interface (cp)
  (let ((guid (make-guid-data)))
    (base-connection-point-interface cp guid)
    (make-guid guid)))
```

95b ⟨*connection point functions* 94f⟩+≡ (92c) ◁95a 95c▷
```
(wrap:std-com-function connection-point-advise "IConnectionPoint_Advise"
                 ((:cptr "IConnectionPoint") (:cptr "AutoServer")
                  (:value dword)))
```

95c ⟨*connection point functions* 94f⟩+≡ (92c) ◁95b 95d▷
```
(wrap:std-com-function connection-point-unadvise "IConnectionPoint_Unadvise"
                       ((:cptr "IConnectionPoint") dword))
```

95d ⟨*connection point functions* 94f⟩+≡ (92c) ◁95c 95e▷
```
;;GetConnectionPointContainer(IConnectionPointContainer **c)
```

95e ⟨*connection point functions* 94f⟩+≡ (92c) ◁95d 95f▷
```
(defmacro with-connection-point ((cp object guid) &body body)
  (let ((cpcsym (gensym))
        (cpsym (gensym)))
    `(with-connection-point-container (,cpcsym ,object)
       (let ((,cpsym (connection-point-container-find ,cpcsym
                                                        (guid-data ,guid))))
         (when (null ,cpsym) (error "can't find connection point"))
         (unwind-protect
             (let ((,cp ,cpsym)) ,@body)
           (release-connection-point ,cpsym))))))
```

95f ⟨*connection point functions* 94f⟩+≡ (92c) ◁95e 95h▷
```
(defun connect-event-listener (object server guid)
  (with-connection-point (cp object guid)
    (let ((cookie (connection-point-advise cp server)))
      (register-event-handler object cookie guid)
      cookie)))
```

95g ⟨*event handling exports* 94d⟩+≡ (92c) ◁94d 95i▷
```
connect-event-listener
```

95h ⟨*connection point functions* 94f⟩+≡ (92c) ◁95f
```
(defun disconnect-event-listener (object cookie guid)
  (with-connection-point (cp object guid)
    (unregister-event-handler object cookie guid)
    (connection-point-unadvise cp cookie)))
```

95i ⟨*event handling exports* 94d⟩+≡ (92c) ◁95g
```
disconnect-event-listener
```

6.2.4 Connection Point Registration

When the source object is garbage-collected, all its listeners are released. Listeners are maintained in a weak hash table, implemented by hashing on the object address. Each entry maintans a weak reference back to the object.

95j ⟨*connection point registration* 95j⟩≡ (92c) 96a ▷

```
(defstruct (event-listener-entry
            (:constructor make-event-listener-entry (object-box cookie guid))
            (:type list))
  object-box cookie guid)
```

96a ⟨*connection point registration* 95j⟩+≡ (92c) ◁ 95j 96b ▷

```
(defparameter *event-listeners* (make-hash-table))
```

96b ⟨*connection point registration* 95j⟩+≡ (92c) ◁ 96a 96c ▷

```
(defun register-event-handler (object cookie guid)
  (let ((key (pointer-address (address-of object)))
        (entry (make-event-listener-entry (make-weak-box object) cookie guid)))
    (debug-print "Registering event handler ... ")
    (push entry (gethash key *event-listeners*))
    (debug-print "done~%")))
```

96c ⟨*connection point registration* 95j⟩+≡ (92c) ◁ 96b 96d ▷

```
(defun unregister-event-handler (object cookie guid)
  (flet ((entry-match (x)
           (and (eq guid (event-listener-entry-guid x))
                (= cookie (event-listener-entry-cookie x)))))
    (let* ((key (pointer-address (address-of object)))
           (entries (gethash key *event-listeners*)))
      (debug-print "Unregistering event handler ... ")
      (let ((entries (remove-if #'entry-match entries)))
        (if entries
            (setf (gethash key *event-listeners*) entries)
          (remhash key *event-listeners*)))
      (debug-print "done~%"))))
```

96d ⟨*connection point registration* 95j⟩+≡ (92c) ◁ 96c 96e ▷

```
(defun disconnect-event-listener-entry (entry)
  (ignore-errors
   (let* ((object-box (event-listener-entry-object-box entry))
          (object (weak-box-value object-box)))
     (when object
       (let ((cookie (event-listener-entry-cookie entry))
             (guid (event-listener-entry-guid entry)))
         (disconnect-event-listener object cookie guid))))))
```

96e ⟨*connection point registration* 95j⟩+≡ (92c) ◁ 96d 96f ▷

```
(defun disconnect-all-event-listeners ()
  (maphash (lambda (key entries)
             (declare (ignore key))
             (dolist (entry entries)
               (disconnect-event-listener-entry entry)))
           *event-listeners*))
```

96f ⟨*connection point registration* 95j⟩+≡ (92c) ◁ 96e

```
(defun disconnect-event-listeners (object)
  (let ((key (pointer-address (address-of object))))
    (dolist (entry (gethash key *event-listeners*))
      (disconnect-event-listener-entry entry))))
```

97a ⟨*test* 47c⟩+≡ ◁92b 97b▷

```
(setf x (get-active-object "Excel.application"))
(setf s (make-auto-server #'sample-getids #'sample-invoke))
(setf eguid (first (get-source-interfaces x)))
(setf c (connect-event-listener x (auto-server-pointer s) eguid))
(setf i (first (get-source-infos x)))
(disconnect-event-listener x c eguid)

**** need to cache info, table mapping memid's to names.
**** are types available too?? useful??
**** disconect on gc of proxy; disconnect on exit
**** cache proxy in weak reference
```

97b ⟨*test* 47c⟩+≡ ◁97a 105c▷

```
(defun event-getgids (&rest args) (apply #'sample-getgids args))

(defun itypeinfo-names-for-memid (info memid)
  (dotimes (i (itypeinfo-function-count info))
    (with-funcdesc (fd info i)
      (when (= memid (funcdesc-memid fd))
        (return (itypeinfo-names info memid (+ (funcdesc-cparams fd) 1)))))))

(defun event-invoke (object dispid lcid flags args keys value)
  (let ((names (itypeinfo-names-for-memid i dispid)))
    (format t "Invoking method ~d = ~a~%" dispid names)
    (format t "Args: ~s~%Named args: ~s~%" args keys)
    0))

(setf x (get-active-object "Excel.application"))
(setf s (make-auto-server 'event-getids 'event-invoke))
(setf eguid (first (get-source-interfaces x)))
(setf c (connect-event-listener x (auto-server-pointer s) eguid))
(setf i (itypelib-type-info-of-guid (itypeinfo-type-lib (idispatch-type-info x)) eguid))
(disconnect-event-listener x c eguid)
```

6.3 Class Factories

This section implements a class factory interface. Class factories can be registered, and are then called by `CoCreateInstance` to produce a server object.

97c ⟨*COM class factories* 97c⟩≡ (81a)

```
(export '(⟨class factory exports 98a⟩))
⟨class factory constant wrappers 101g⟩
⟨class factory function wrappers 99a⟩
⟨class factory lisp interface 97d⟩
```

6.3.1 Class Factory Structure

At the Lisp level class factories are represented as structures. A class factory structure contains an optional `ProgID`, a creation callback that is used to actually create objects, a `GUID`, and a slot for a registration cookie. Including the generic server structure also brings in a slot for a pointer for the internal implementation.

97d ⟨*class factory lisp interface* 97d⟩≡ (97c) 98b ▷
```
(defstruct (class-factory
            (:include generic-server)
            (:constructor new-class-factory (creator pointer)))
  creator)
```
The structure name is exported for use in typing expressions.

98a ⟨*class factory exports* 98a⟩≡ (97c) 98c ▷
```
class-factory
```
The public constructor for a class factory structure requires one argument. This argument is a creator function, a function of no arguments that is called to create a server object.

98b ⟨*class factory lisp interface* 97d⟩+≡ (97c) ◁ 97d 99c ▷
```
(defun make-class-factory (creator)
  (let* ((pointer (base-make-class-factory))
         (factory (new-class-factory creator pointer)))
    (system:cptr-protect pointer factory)
    (init-class-factory pointer factory)
    factory))
```
98c ⟨*class factory exports* 98a⟩+≡ (97c) ◁ 98a 101f ▷
```
make-class-factory
```

6.3.2 IClassFactory Interface

The internal representation of a class factory corresponds to the `IClassFactory` interface. The class factory is represented by a C structure.

98d ⟨*class factory support declarations* 98d⟩≡ (112b) 98e ▷
```
typedef struct tagClassFactory ClassFactory;
```
The Vtbl structure for the factory is given by

98e ⟨*class factory support declarations* 98d⟩+≡ (112b) ◁ 98d 98f ▷
```
typedef struct {
  ⟨IUnknown Vtbl entries 82a⟩
  HRESULT (STDMETHODCALLTYPE *CreateInstance)(ClassFactory *This,
                                              IUnknown *outer,
                                              REFIID iid, void **ppv);
  HRESULT (STDMETHODCALLTYPE *LockServer)(ClassFactory *This, BOOL lock);
} ClassFactoryVtbl;
```
The `tagClassFactory` structure is given by

98f ⟨*class factory support declarations* 98d⟩+≡ (112b) ◁ 98e 99b ▷
```
struct tagClassFactory {
  ClassFactoryVtbl *lpVtbl;
  ⟨generic interface fields 82b⟩
};
```
The class factory `QueryInterface` method signals an error unless it is asked for the `IUnknown` or the `IClassFactory` interface.

98g ⟨*class factory support functions* 98g⟩≡ (112c) 99d ▷
```
static STDMETHODIMP
ClassFactory_QueryInterface(GenericServer *this, REFIID riid, void **ppv)
{
  if (IsEqualIID(riid, &IID_IUnknown) ||
```

```
      IsEqualIID(riid, &IID_IClassFactory)) {
    IUnknown_AddRef((IUnknown *) this);
    *ppv = this;
    return S_OK;
  }
  else {
    *ppv = NULL;
    return E_NOINTERFACE;
  }
}
```

The `CreateInstance` method uses a Lisp callback to create the server object. The callback returns a pointer to a generic server object. Errors and non-local exits are trapped; if either occurs the returned result will be `NULL`.

99a ⟨*class factory function wrappers* 99a⟩≡ (97c) 99e ▷

```
(wrap:c-callback "FactoryCreateInstance" factory-create-instance
                 (:lval) (:cptr "GenericServer")
                 :static nil
                 :interrupts-allowed nil
                 :trap-exits "NULL")
```

99b ⟨*class factory support declarations* 98d⟩+≡ (112b) ◁ 98f 100a ▷

```
GenericServer *FactoryCreateInstance(LVAL object);
```

The callback calls the creator function and casts the pointer of the server object to a generic server pointer.

99c ⟨*class factory lisp interface* 97d⟩+≡ (97c) ◁ 98b 100c ▷

```
(defun factory-create-instance (factory)
  (let ((server (funcall (class-factory-creator factory))))
    (cast-generic-server (generic-server-pointer server))))
```

The `CreateInstance` method uses the callback to create the server and then calls `QueryInterface` to get the right interface pointer and increment the reference count. The server object will not be reachable from the object graph, but it will be protected from GC by the `AddRef` method. Aggregation is not supported, so an error is signaled if it is requested.

99d ⟨*class factory support functions* 98g⟩+≡ (112c) ◁ 98g 100b ▷

```
static STDMETHODIMP
ClassFactory_CreateInstance(ClassFactory *this, IUnknown *outer,
                            REFIID iid, void **ppv)
{
  if (outer != NULL) {
    *ppv = NULL;
    return CLASS_E_NOAGGREGATION;
  }
  else {
    AutoServer *server = FactoryCreateInstance(this->object);
    if (server == NULL) return E_UNEXPECTED;
    else return IUnknown_QueryInterface(server, iid, ppv);
  }
}
```

The second `IClassFactory` method, `LockServer`, is also implemented as a callback. This is mainly because I can't figure out yet what to do with it.

99e ⟨*class factory function wrappers* 99a⟩+≡ (97c) ◁99a 100g ▷
```
(wrap:c-callback "FactoryLockServer" factory-lock-server (:lval bool)
        hresult
        :static nil
        :interrupts-allowed nil
        :trap-exits "E_UNEXPECTED")
```

100a ⟨*class factory support declarations* 98d⟩+≡ (112b) ◁99b 100f ▷
```
HRESULT FactoryLockServer(LVAL object, BOOL lock);
```

100b ⟨*class factory support functions* 98g⟩+≡ (112c) ◁99d 100d ▷
```
static STDMETHODIMP ClassFactory_LockServer(ClassFactory *this, BOOL lock)
{
  return FactoryLockServer(this->object, lock);
}
```

The initial callback implementation just prints a message to indicate it was called.

100c ⟨*class factory lisp interface* 97d⟩+≡ (97c) ◁99c 101a ▷
```
(defun factory-lock-server (object lock)
  (system:without-exits
   (let ((message (if (= lock 0) "unlocking server" "locking server")))
     (debug-print "~a~%" message))))
```

The actual Vtbl for a class factory is given by

100d ⟨*class factory support functions* 98g⟩+≡ (112c) ◁100b 100e ▷
```
static ClassFactoryVtbl my_FactoryVtbl = {
  ClassFactory_QueryInterface,
  Generic_AddRef,
  Generic_Release,
  ClassFactory_CreateInstance,
  ClassFactory_LockServer
};
```

The initialization function installs the Vtbl and initializes the generic data fields.

100e ⟨*class factory support functions* 98g⟩+≡ (112c) ◁100d
```
void InitializeClassFactory(ClassFactory *fac, LVAL object)
{
  fac->lpVtbl = &my_FactoryVtbl;
  fac->count = 0;
  fac->object = object;
  fac->isClassFac = TRUE;
}
```

100f ⟨*class factory support declarations* 98d⟩+≡ (112b) ◁100a
```
void InitializeClassFactory(ClassFactory *fac, LVAL object);
```

100g ⟨*class factory function wrappers* 99a⟩+≡ (97c) ◁99e 100h ▷
```
(wrap:c-function init-class-factory "InitializeClassFactory"
                 ((:cptr "ClassFactory") :lval) :void)
```

The creator of the internal representation of a class factory is defined with a pointer wrapper.

100h ⟨*class factory function wrappers* 99a⟩+≡ (97c) ◁100g 102b ▷
```
(wrap:c-pointer "ClassFactory" (:make base-make-class-factory))
```

6.3.3 Class Factory Registration

The registration function takes a factory structure and several keyword arguments. The keyword arguments can be used to specify the context and the type of server. An additional keyword can be used to specify that the server should be created suspended; this seems to only make sense if DCOM is available; the argument is ignored if DCOM is not available.

101a ⟨*class factory lisp interface* 97d⟩+≡ (97c) ◁100c 101b▷
```
(defparameter *class-factories* nil)
```

101b ⟨*class factory lisp interface* 97d⟩+≡ (97c) ◁101a 101c▷
```
(defun find-class-factory-entry (clsid)
  (find clsid *class-factories* :key #'registration-entry-clsid))
```

101c ⟨*class factory lisp interface* 97d⟩+≡ (97c) ◁101b 101d▷
```
(defun find-class-factory-entries (factory)
  (let ((val nil))
    (dolist (entry *class-factories* val)
      (when (eq factory (registration-entry-server entry)) (push entry val)))))
```

101d ⟨*class factory lisp interface* 97d⟩+≡ (97c) ◁101c 101e▷
```
(defun enter-class-factory (clsid server cookie)
  (when (find-class-factory-entry clsid)
    (error "already have a class factory for CLSID ~a" clsid))
  (push (make-registration-entry clsid server cookie) *class-factories*))

(defun remove-class-factory-entry (entry)
  (setf *class-factories* (remove entry *class-factories*)))
```

101e ⟨*class factory lisp interface* 97d⟩+≡ (97c) ◁101d 102c▷
```
(defun register-class-factory (fac cls-spec &key
                                  (context :server)
                                  (type :multiple)
                                  suspended)
  (initialize-com)
  (let* ((clsid (find-clsid cls-spec))
         (entry (find-class-factory-entry clsid)))
    (when entry
      (revoke-class-factory (registration-entry-server entry) clsid))
    (let* ((pointer (class-factory-pointer fac))
           (gd (guid-data clsid))
           (cntxt ⟨translate keyword to CLSCTX value 39c⟩)
           (tflag ⟨translate keyword to REGCLS value 102a⟩)
           (flags (if (and suspended (dcom-available))
                      (logior REGCLS_SUSPENDED tflag)
                    tflag))
           (cookie (base-register-class-factory gd pointer cntxt flags)))
      (enter-class-factory clsid fac cookie)
      t)))
```

101f ⟨*class factory exports* 98a⟩+≡ (97c) ◁98c 102d▷
```
register-class-factory
```

The registration flags allow the type of server to be specified using the constants

101g ⟨*class factory constant wrappers* 101g⟩≡ (97c)
```
(wrap:c-constant REGCLS_SINGLEUSE "REGCLS_SINGLEUSE" :unsigned)
(wrap:c-constant REGCLS_MULTIPLEUSE "REGCLS_MULTIPLEUSE" :unsigned)
(wrap:c-constant REGCLS_MULTI_SEPARATE "REGCLS_MULTI_SEPARATE" :unsigned)
(wrap:c-constant REGCLS_SUSPENDED "REGCLS_SUSPENDED" :unsigned)
```

The registration function uses a keyword to specify the type; it can be `single`, `:multiple`, or `:multi-single`.

102a ⟨*translate keyword to* `REGCLS` *value* 102a⟩≡ (101e)
```
(ecase type
       (:single REGCLS_SINGLEUSE)
       (:multiple REGCLS_MULTIPLEUSE)
       (:multi-separate REGCLS_MULTI_SEPARATE))
```

The internal registration function `CoRegisterClassObject` is wrapped as

102b ⟨*class factory function wrappers* 99a⟩+≡ (97c) ◁100h 102e▷
```
(wrap:std-com-function base-register-class-factory "CoRegisterClassObject"
                 ((:cptr "GUID") (:cptr "ClassFactory") dword dword
                  (:value dword)))
```

A registered server is revoked by calling `revoke-class-factory`.

102c ⟨*class factory lisp interface* 97d⟩+≡ (97c) ◁101e
```
(defun revoke-class-factory (factory &optional cls-spec)
  (flet ((revoke-entry (entry)
           (remove-class-factory-entry entry)
           (base-revoke-class-factory (registration-entry-cookie entry))))
    (if cls-spec
        (let ((entry (find-class-factory-entry (find-clsid cls-spec))))
          (when (and entry
                     (or (null factory)
                         (eq factory (registration-entry-server entry))))
            (revoke-entry entry)))
      (dolist (entry (find-class-factory-entries factory))
        (revoke-entry entry)))))

(wrap:std-com-function base-revoke-class-factory "CoRevokeClassObject"
                       (dword))
```

102d ⟨*class factory exports* 98a⟩+≡ (97c) ◁101f 103a▷
```
revoke-class-factory
```

When DCOM is available is is possible to create factories suspended and to then call `CoResumeClassObjects` once all factories are registered. All currently registered factories can be suspended by calling `CoSuspendClassObjects`. These functions do not exist if DCOM is not available, so they are accessed through function pointers that are looked up in the appropriate DLL. The Lisp interfaces do nothing when these functions are not available.

102e ⟨*class factory function wrappers* 99a⟩+≡ (97c) ◁102b
```
(wrap:c-lines "
static HRESULT ResumeClassObjects(void)
{
  if (pCoResumeClassObjects != NULL) return pCoResumeClassObjects();
  else return S_OK;
}
static HRESULT SuspendClassObjects(void)
{
```

```
    if (pCoSuspendClassObjects != NULL) return pCoSuspendClassObjects();
    else return S_OK;
  }")

  (wrap:std-com-function resume-class-factories "ResumeClassObjects" ())
  (wrap:std-com-function suspend-class-factories "SuspendClassObjects" ())
```

103a ⟨*class factory exports* 98a⟩+≡ (97c) ◁102d

```
resume-class-factories
suspend-class-factories
```

All class factories can be revoked by calling `revoke-all-class-factories`. This is used when COM is uninitialized.

103b ⟨*server lisp interface* 81c⟩+≡ (81b) ◁91f

```
(defun revoke-all-class-factories ()
  (dolist (entry *class-factories*)
    (let ((server (registration-entry-server entry))
          (clsid (registration-entry-clsid entry)))
      (revoke-class-factory server clsid))))
```

6.4 Higher Level Server Interface

103c ⟨*higher level server interface* 103c⟩≡ (81a)

```
(export '(⟨higher level server exports 103f⟩))
⟨higher level servers 103d⟩
⟨higher level active objects 105b⟩
⟨higher level class factories 105d⟩
⟨higher level event handlers 106b⟩
⟨higher level server registration 107a⟩
```

6.4.1 COM and Automation Servers

**** need to redefine `object-to-variant`.

103d ⟨*higher level servers* 103d⟩≡ (103c) 103e▷

```
(defun object-to-variant (object type pvar)
  (struct-to-variant (send object :server) type pvar nil))
```

103e ⟨*higher level servers* 103d⟩+≡ (103c) ◁103d 103g▷

```
(defproto com-server)
(defproto auto-server '(methods server-name server) () com-server)
```

103f ⟨*higher level server exports* 103f⟩≡ (103c) 107b▷

```
com-server auto-server
```

103g ⟨*higher level servers* 103d⟩+≡ (103c) ◁103e 103h▷

```
(defmeth auto-server :auto-methods () (slot-value 'methods))
(defmeth auto-server :server-name () (slot-value 'server-name))
(defmeth auto-server :server () (slot-value 'server))
```

103h ⟨*higher level servers* 103d⟩+≡ (103c) ◁103g 104a▷

```
(defmeth auto-server :isnew (&optional (name "Xlisp-Stat"))
  (setf (slot-value 'server-name) "Xlisp-Stat")
  (setf (slot-value 'server) (send self :make-server)))
```

104a ⟨*higher level servers* 103d⟩+≡ (103c) ◁ 103h 104b ▷

```
(defmeth auto-server :add-auto-method (com-name name &optional (for-value t))
  (let ((entry (find com-name (slot-value 'methods) :key #'first)))
    (unless entry
      (setf entry (list com-name :method nil nil))
      (setf (slot-value 'methods)
            (concatenate 'vector (slot-value 'methods) (list entry))))
    (setf (third entry) name)
    (setf (fourth entry) for-value)))
```

104b ⟨*higher level servers* 103d⟩+≡ (103c) ◁ 104a 104c ▷

```
(defmeth auto-server :add-auto-property (com-name get-name &optional set-name)
  (let ((entry (find com-name (slot-value 'methods) :key #'first)))
    (unless entry
      (setf entry (list com-name :property nil nil))
      (setf (slot-value 'methods)
            (concatenate 'vector (slot-value 'methods) (list entry))))
    (setf (third entry) get-name)
    (setf (fourth entry) set-name)))
```

104c ⟨*higher level servers* 103d⟩+≡ (103c) ◁ 104b 104d ▷

```
(send auto-server :add-auto-property :name :server-name)
```

104d ⟨*higher level servers* 103d⟩+≡ (103c) ◁ 104c 104e ▷

```
(defconstant keyword-package (find-package "KEYWORD"))

(defmeth auto-server :get-ids-of-names (object name keys lcid)
  (declare (ignore object lcid))
  ;;**** Error if keys requested; maybe also check lcid?
  ;;**** handle not found error properly
  (debug-print "using method~%")
  (let ((methods (send self :auto-methods))
        (ksym (intern (string-upcase name) keyword-package)))
    (position ksym methods :key #'first)))
```

104e ⟨*higher level servers* 103d⟩+≡ (103c) ◁ 104d 105a ▷

```
(defmeth auto-server :invoke (object dispid lcid flags args keys value)
  ;;check for keywords
  (debug-print "invoking method~%")
  (let ((entry (aref (send self :auto-methods) dispid)))
    (cond
     ((/= (logand DISPATCH_PROPERTYPUTREF flags) 0)
      (error "PUTREF not supported"))
     ((/= (logand DISPATCH_PROPERTYPUT flags) 0)
      (unless (eq (second entry) :property)
        (error "~a is not a property" (first entry)))
      (let ((meth (fourth entry)))
        (unless meth
          (error "property ~a is read-only" (first entry)))
        (apply #'send self meth args)))
     (t;;**** more careful check?
      (let ((meth (third entry)))
        (unless meth
          (if (eq (second entry) :property)
```

```
                (error "property ~a is not readable" (first entry))
              (error "method has no definition")))
          (apply #'send self meth args))))))
```

105a ⟨*higher level servers* 103d⟩+≡ (103c) ◁104e

```
  (defmeth auto-server :make-server ()
    (flet ((getids (object name keys lcid)
             (send self :get-ids-of-names object name keys lcid))
           (invoke (object dispid lcid flags args keys value)
             (send self :invoke object dispid lcid flags args keys value)))
      (make-auto-server #'getids #'invoke (send self :server-name))))
```

6.4.2 Active Objects

**** should server creation be deferred until it is needed? Should it be handled by auto-server?

105b ⟨*higher level active objects* 105b⟩≡ (103c)

```
  (defmeth auto-server :register-active-object (cls-spec)
    (register-active-object (slot-value 'server) cls-spec))

  (defmeth auto-server :revoke-active-object (&optional cls-spec)
    (revoke-active-object (slot-value 'server) cls-spec))
```

105c ⟨*test* 47c⟩+≡ ◁97b 106a▷

```
  (defproto sample-server '(x y) () auto-server)
  (defmeth sample-server :isnew ()
    (call-next-method)
    (setf (slot-value 'x) 1)
    (setf (slot-value 'y) 2))

  (defmeth sample-server :get-x () (slot-value 'x))
  (defmeth sample-server :set-x (v) (setf (slot-value 'x) v))
  (send sample-server :add-auto-property :x :get-x :set-x)

  (defmeth sample-server :get-y () (slot-value 'y))
  (send sample-server :add-auto-property :y :get-y)

  (defmeth sample-server :xpy (&optional (z 0))
    (+ (slot-value 'x) (slot-value 'y) z))
  (send sample-server :add-auto-method :xpy :xpy)

  (setf s (send sample-server :new))
  (send s :register-active-object "{FB4C2CC0-60EF-11D3-8E98-444553540000}")
  (setf x (get-active-object "{FB4C2CC0-60EF-11D3-8E98-444553540000}"))
  (property x :x)
  (invoke x :xpy)
  (invoke x :xpy 4)
  (send s :revoke-active-object)
```

6.4.3 Class Factories

105d ⟨*higher level class factories* 105d⟩≡ (103c)

```
  (defproto class-factory '(prototype singleton factory))
```

```
(defmeth class-factory :isnew (proto &key singleton)
  (setf (slot-value 'prototype) proto)
  (setf (slot-value 'singleton) singleton)
  (let ((class-fun (if singleton
                       (lambda () (send proto :server))
                     (lambda () (send (send proto :new) :server)))))
    (setf (slot-value 'factory) (make-class-factory class-fun))))

(defmeth class-factory :register (cls-spec)
  (register-class-factory (slot-value 'factory) cls-spec))

(defmeth class-factory :revoke (&optional cls-spec)
  (revoke-class-factory (slot-value 'factory) cls-spec))
```

106a ⟨*test* 47c⟩+≡ ◁ 105c

```
(setf sample-class-factory (send class-factory :new sample-server))
(send sample-class-factory :register "XlispStat.application")
(setf x (create-object "XlispStat.application"))
(property x :x)
(send sample-class-factory :revoke)
```

6.4.4 Event Handlers

106b ⟨*higher level event handlers* 106b⟩≡ (103c)

```
(defproto event-server '(source cookie eguid info) () auto-server)
(export 'event-server)

(defmeth event-server :isnew (source)
  (let* ((eguid (first (get-source-interfaces source)))
         (lib (itypeinfo-type-lib (idispatch-type-info source)))
         (info (itypelib-type-info-of-guid lib eguid)))
    (setf (slot-value 'source) source)
    (setf (slot-value 'eguid) eguid)
    (setf (slot-value 'info) info)))

(defmeth event-server :get-ids-of-names (object name keys lcid)
  (error "Need to override this properly"))

(defun itypeinfo-names-for-memid (info memid)
  (dotimes (i (itypeinfo-function-count info))
    (with-funcdesc (fd info i)
      (when (= memid (funcdesc-memid fd))
        (return (itypeinfo-names info memid (+ (funcdesc-cparams fd) 1)))))))

;;**** need to cache this stuff
(defmeth event-server :invoke (object dispid lcid flags args keys value)
  (let* ((info (slot-value 'info))
         (names (itypeinfo-names-for-memid info dispid)))
      (debug-print "Invoking method ~d = ~a~%" dispid names)
      (debug-print "Args: ~s~%Named args: ~s~%" args keys)
      (let ((meth (intern (string-upcase (first names)) keyword-package)))
        (when (send self :has-method meth)
```

```
          (apply #'send self meth args)))
        0))

  (defmeth event-server :connect ()
    (when (slot-value 'cookie) (error "already connected"))
    (let* ((source (slot-value 'source))
           (eguid (slot-value 'eguid))
           (s (send self :make-server))
           (ptr (auto-server-pointer s)))
      (setf (slot-value 'cookie) (connect-event-listener source ptr eguid))))

  (defmeth event-server :disconnect ()
    (let ((source (slot-value 'source))
          (cookie (slot-value 'cookie))
          (eguid (slot-value 'eguid)))
      (unless cookie (error "not connected"))
      (setf (slot-value 'cookie) nil)
      (disconnect-event-listener source cookie eguid)))
```

6.4.5 Server Registration

This registration process is based on the C++ `RegisterServer` and `UnregisterServer` functions provided in [2].

The function `register-com-server` performs the registry operations needed to register the server. Its arguments are a string with the command to start the server, a `CLSID` string, a "friendly name" string for labeling the registered class in browsers, the `ProgID` string, and a version-independent `ProgID` string.

107a ⟨*higher level server registration* 107a⟩≡ (103c) 107c ▹

```
  (defun register-com-server (program clsid friendly-name progid vi-progid)
    (flet ((set-key-and-value (key subkey value)
             (let* ((keyname (if subkey (format nil "~a\\~a" key subkey) key))
                    (newkey (win32:reg-create-subkey win32:hkey-classes-root
                                                     keyname)))
               (win32:reg-set-value newkey "" value)
               (win32::reg-close-key newkey))))
      (let ((clsid-key (format nil "CLSID\\~a" clsid)))
        (set-key-and-value clsid-key nil friendly-name)
        (set-key-and-value clsid-key "LocalServer32" program) ;**** args
        (set-key-and-value clsid-key "ProgID" progid)
        (set-key-and-value clsid-key "VersionIndependentProgID" vi-progid)

        (set-key-and-value vi-progid nil friendly-name)
        (set-key-and-value vi-progid "CLSID" clsid)
        (set-key-and-value vi-progid "CurVer" progid)

        (set-key-and-value progid nil friendly-name)
        (set-key-and-value progid "CLSID" clsid))))
```

107b ⟨*higher level server exports* 103f⟩+≡ (103c) ◃ 103f 108a ▹

```
  register-com-server
```

The function `unregister-com-server` removes a server registration from the registry. Its arguments are the `CLSID`, the `ProgID` and the version-independent `ProgID`'s used to register the server. All subkeys of the corresponding keys are deleted recursively.

107c ⟨*higher level server registration* 107a⟩+≡ (103c) ◁107a 108b▷

```
(defun unregister-com-server (clsid progid vi-progid)
  (labels ((recursive-delete-key (parent child)
             (handler-case
              (let ((key (win32:reg-open-key parent child)))
                (dolist (n (win32:reg-subkey-names key))
                  (recursive-delete-key key n))
                (win32:reg-close-key key)
                (win32:reg-delete-subkey parent child))
              (error (c) (format *debug-io* "error processing ~a:~a: ~a~%"
                                 parent child c)))))
    (let ((clsid-key (format nil "CLSID\\~a" clsid)))
      (ignore-errors
       (recursive-delete-key win32:hkey-classes-root clsid-key))
      (ignore-errors
       (recursive-delete-key win32:hkey-classes-root vi-progid))
      (ignore-errors
       (recursive-delete-key win32:hkey-classes-root progid)))))
```

108a ⟨*higher level server exports* 103f⟩+≡ (103c) ◁107b

```
unregister-com-server
```

The loop to do this uses a list of subkeys since deleting seem to confuse the mapping function.

For debugging purposes, `lisp-stat-server-info` prints the registration information for the Lisp-Stat server.

108b ⟨*higher level server registration* 107a⟩+≡ (103c) ◁107c

```
(defun lisp-stat-server-info (&optional (stream *standard-output*))
  (labels ((recursive-write-key (parent child)
             (let ((key (win32:reg-open-key parent child)))
               (format stream "~a: ~a~%" key (win32:reg-query-value key))
               (win32:reg-map-subkey-names
                (lambda (n) (recursive-write-key key n))
                key)
               (win32:reg-close-key key))))
    (let ((progid "XlispStat.application.1")
          (vi-progid "XlispStat.application")
          (clsid "{FB4C2CC0-60EF-11D3-8E98-444553540000}"))
      (let ((clsid-key (format nil "CLSID\\~a" clsid)))
        (recursive-write-key win32:hkey-classes-root clsid-key)
        (recursive-write-key win32:hkey-classes-root vi-progid)
        (recursive-write-key win32:hkey-classes-root progid)))))
```

7 Support Utilities

108c ⟨*support utilities* 108c⟩≡ (25b)

⟨*pointer casting* 108d⟩
⟨*wide string encoding/decoding* 109a⟩
⟨*checking for DCOM* 110h⟩

7.1 Pointer Casting

108d ⟨*pointer casting* 108d⟩≡ (108c)

```
(wrap:c-lines "
static LVAL cast_pointer(void)
{
  return xlw_cast_cptr(xlgetarg());
}")
(wrap:c-subr base-cast-pointer "cast_pointer")
(wrap:c-function lookup-pointer-type "xlw_lookup_type" (:string) :lval)

(defun make-pointer-caster (tname)
  (let ((type (lookup-pointer-type tname)))
    (lambda (pointer) (base-cast-pointer type pointer))))
```

7.2 Wide String Encoding and Decoding

109a ⟨*wide string encoding/decoding* 109a⟩≡ (108c) 109b ▷

```
(wrap:c-pointer "WCHAR" (:make make-wide-string))
```

109b ⟨*wide string encoding/decoding* 109a⟩+≡ (108c) ◁ 109a 109c ▷

```
;;***** fix wrapper documentation--defualt is NULL is NOT allowed
(wrap:c-function multi-byte-to-wide-char "MultiByteToWideChar"
                 (:unsigned dword :string :integer (:cptr "WCHAR" t)
                            :integer)
                 :integer)
```

109c ⟨*wide string encoding/decoding* 109a⟩+≡ (108c) ◁ 109b 109d ▷

```
(wrap:c-function wide-char-to-multi-byte "WideCharToMultiByte"
                 (:unsigned dword (:cptr "WCHAR") :integer :string :integer
                             :string (:cptr "BOOL" t))
                 :integer)
```

109d ⟨*wide string encoding/decoding* 109a⟩+≡ (108c) ◁ 109c 109f ▷

```
(defun string-to-wide-string (string)
  (let ((res (multi-byte-to-wide-char CP_ACP 0 string -1 nil 0)))
    (if (= res 0)
        (error "can't convert ~s to wide string" string)
      (let ((wstring (make-wide-string res)))
        (multi-byte-to-wide-char CP_ACP 0 string -1 wstring res)
        wstring))))
```

109e ⟨*basic constants* 109e⟩≡ (25b)

```
(wrap:c-constant CP_ACP "CP_ACP" :unsigned)
```

109f ⟨*wide string encoding/decoding* 109a⟩+≡ (108c) ◁ 109d

```
;;***** Fix wrappers to :string or (:string t) is allowed--NULL
;;***** Fix wrappers to test for string in :string!!
;;***** change "" to nil when wrappers are fixed
(defun wide-string-to-string (wstring)
  (let ((res (wide-char-to-multi-byte CP_ACP 0 wstring -1 "" 0 "" nil)))
    (if (= res 0)
        (error "can't convert from wide string")
      (let ((string (make-string (- res 1))))
        (wide-char-to-multi-byte CP_ACP 0 wstring -1 string res "" nil)
        string))))
```

7.3 Unwind-Protect Macros

110a ⟨*unwind-protect macros* 110a⟩≡ (112b) 110b ▷

```
#define BEGIN_PROTECT \
{ \
  CONTEXT _unwind_cntxt, *_unwind_target; \
  int _unwind_mask, _unwinding_; \
  LVAL _unwind_value; \
  xlbegin(&_unwind_cntxt,CF_UNWIND,NIL); \
  if (setjmp(_unwind_cntxt.c_jmpbuf)) { \
    _unwinding_ = TRUE; \
    _unwind_target = xltarget; \
    _unwind_mask = xlmask; \
    _unwind_value = xlvalue; \
  } \
  else { \
    _unwinding_ = FALSE; \
    _unwind_target = NULL; \
    _unwind_mask = 0; \
    _unwind_value = NIL; \
    {
```

110b ⟨*unwind-protect macros* 110a⟩+≡ (112b) ◁ 110a 110c ▷

```
#define BEGIN_CLEANUP \
  } \
} \
xlend(&_unwind_cntxt); \
{
```

110c ⟨*unwind-protect macros* 110a⟩+≡ (112b) ◁ 110b 110d ▷

```
#define END_PROTECT \
  } \
  if (_unwinding_) xljump(_unwind_target, _unwind_mask, _unwind_value); \
}
```

110d ⟨*unwind-protect macros* 110a⟩+≡ (112b) ◁ 110c

```
#define UNWINDING _unwinding_
```

110e ⟨*begin protect* 110e⟩≡

```
BEGIN_PROTECT
```

110f ⟨*begin cleanup* 110f⟩≡

```
BEGIN_CLEANUP
```

110g ⟨*end protect* 110g⟩≡

```
END_PROTECT
```

7.4 Checking For DCOM

**** put this someplace alse? It is useful to be able to determnine from the Lisp level whether `DCOM` is available. **** it ought to be possible to make this a constant–but only per-session, not per-workspace

110h ⟨*checking for DCOM* 110h⟩≡ (108c)
```
;;**** allow boolean type
(wrap:c-lines "
static BOOL IsDcomAvailable(void)
{
  return pCoCreateInstanceEx != NULL ? TRUE : FALSE;
}")
(wrap:c-function dcom-available "IsDcomAvailable" () :bool)
```

7.5 Miscellaneous Stuff

This is a `.reg` file for registering an `XlispStat.application` server. Eventually this should be replaced by an internal machanism that does the registration and unregistration when the server is called with the appropriate command line arguments.

For the moment, the executable explicitly loads the files it needs. THis should be handled more cleanly eventually.

111a ⟨*wxls32.reg* 111a⟩≡
```
REGEDIT
; This .REG file may be used by your SETUP program.

HKEY_CLASSES_ROOT\XlispStat.application = XlispStat.application
HKEY_CLASSES_ROOT\XlispStat.application\CLSID = ⟨XlispStat CLSID 111b⟩
HKEY_CLASSES_ROOT\CLSID\⟨XlispStat CLSID 111b⟩ = XlispStat.application
HKEY_CLASSES_ROOT\CLSID\⟨XlispStat CLSID 111b⟩\LocalServer32 = ⟨XlispStat home 111c⟩\wxls32.exe Autoload\win32com\server
HKEY_CLASSES_ROOT\CLSID\⟨XlispStat CLSID 111b⟩\ProgID = XlispStat.application
```

The CLSID for the `XlispStat.application` ProgID is given by

111b ⟨*XlispStat CLSID* 111b⟩≡ (111a)
```
{FB4C2CC0-60EF-11D3-8E98-444553540000}
```

This should never be changed.

111c ⟨*XlispStat home* 111c⟩≡ (111a)
```
\users\luke\xlispstat
```

111d ⟨*making the* `_autoidx.lsp` *file* 111d⟩≡
```
(defun make-autoidx (file module &rest packages)
  (with-open-file (f file :direction :output)
    (format f "(provide ~s)~%" module)
    (dolist (pack packages)
      (let ((syms nil)
            (funs nil)
            (vars nil))
        (do-external-symbols (s pack)
          (push s syms)
          (when (fboundp s) (push s funs))
          (when (boundp s) (push s vars)))
        (format
         f "(defpackage ~s~@[ (:nicknames~{ ~s~})~]~@[ (:use~{ ~s~})~])~%"
         (package-name pack)
         (package-nicknames pack)
         (mapcar #'package-name (package-use-list pack)))
        (format f "(in-package ~s)~%~%" (package-name pack))
        (format f "(export '(~{~a ~}))~%~%" syms)
```

```
          (format f "(system:define-autoload-module ~s~
                     ~@[~&  (function~{ ~a~})~]~
                     ~@[~&  (variable~{ ~a~})~])~%"
                   module
                   funs
                   vars)))))

  (make-autoidx "_autoidx.lsp" "win32com" "WIN32-COM")
```

8 Files

112a ⟨*win32com.wrp* 112a⟩≡
  ⟨*COM basics* 25b⟩
  ⟨*COM client support* 38a⟩
  ⟨*type libraries and type information* 63d⟩
  ⟨*COM server support* 81a⟩

**** need `-DCOBJMACROS` in `CFLAGS` for wrapper file!! Or maybe way to put stuff in wrapper file ahead of standard include.

112b ⟨*comutil.h* 112b⟩≡

```
#define COBJMACROS
#include <ocidl.h>

⟨error signaling support declarations 33d⟩
⟨variant type conversion declarations 51d⟩
⟨invokation support declarations 46a⟩
⟨type conversion declarations 41a⟩
⟨server support declarations 81e⟩
⟨class factory support declarations 98d⟩
⟨unwind-protect macros 110a⟩

typedef HRESULT STDAPICALLTYPE (*tCoCreateInstanceEx)
     (REFCLSID, IUnknown *, DWORD, COSERVERINFO*, DWORD, MULTI_QI *);
typedef HRESULT STDAPICALLTYPE (*tCoResumeClassObjects)(void);
typedef HRESULT STDAPICALLTYPE (*tCoSuspendClassObjects)(void);
extern tCoCreateInstanceEx pCoCreateInstanceEx;
extern tCoResumeClassObjects pCoResumeClassObjects;
extern tCoSuspendClassObjects pCoSuspendClassObjects;
```

112c ⟨*comutil.c* 112c⟩≡

```
#include "xlshlib.h"
#include "xlwrap.h"
#include "comutil.h"

⟨variant type conversion functions 52a⟩
⟨invokation support functions 44g⟩
⟨server support functions 82e⟩
⟨class factory support functions 98g⟩
```

112d ⟨*dllstub.c* 112d⟩≡

```
#include <windows.h>

extern CRITICAL_SECTION server_protect_cs;

typedef HRESULT STDAPICALLTYPE (*tCoCreateInstanceEx)
     (REFCLSID, IUnknown *, DWORD, COSERVERINFO*, DWORD, MULTI_QI *);
typedef HRESULT STDAPICALLTYPE (*tCoResumeClassObjects)(void);
typedef HRESULT STDAPICALLTYPE (*tCoSuspendClassObjects)(void);

tCoCreateInstanceEx pCoCreateInstanceEx;
tCoResumeClassObjects pCoResumeClassObjects;
tCoSuspendClassObjects pCoSuspendClassObjects;
static HINSTANCE ole32lib;

/**** I seem to get the process calls but not the thread one??*/
int APIENTRY DllMain(HANDLE hdll, DWORD  reason, LPVOID reserved )
{
  switch( reason ) {
  case DLL_THREAD_ATTACH: break;
  case DLL_THREAD_DETACH: break;
  case DLL_PROCESS_ATTACH:
    InitializeCriticalSection(&server_protect_cs);
    ole32lib = LoadLibrary("OLE32");
    if (ole32lib) {
      pCoCreateInstanceEx = (tCoCreateInstanceEx)
        GetProcAddress(ole32lib, "CoCreateInstanceEx");
      pCoResumeClassObjects = (tCoResumeClassObjects)
        GetProcAddress(ole32lib, "CoResumeClassObjects");
      pCoSuspendClassObjects = (tCoSuspendClassObjects)
        GetProcAddress(ole32lib, "CoSuspendClassObjects");
    }
    else {
      pCoCreateInstanceEx = NULL;
      pCoResumeClassObjects = NULL;
      pCoSuspendClassObjects = NULL;
    }
    break;
  case DLL_PROCESS_DETACH:
    if (ole32lib != NULL) FreeLibrary(ole32lib);
    DeleteCriticalSection(&server_protect_cs);
    break;
  }
  return( 1 );
}

/* The Borland entry point. C*/
BOOL APIENTRY DllEntryPoint(HINSTANCE hInst, DWORD reason, LPVOID reserved)
{
  return DllMain(hInst, reason, reserved);
}
```

9 To Do

- **** fix up description environments
- **** test on Win 95/98
- **** make preliminary release
- **** fix use of lambda expression in call handlers (currently assumes apply makes closure–do it explicitly?).
- **** dynamic adding of properties, methods??
- **** don't start DDE server for COM server??
- **** active object for Lisp-Stat server???
- **** scripting host/scripting clients??
- **** test event stuff–check for shutdown errors
- **** test distributed stuff (NT security?)
- **** clean up registration??
- **** make all generic server methods callbacks?
- **** add ignore-errors to active object, class factory cleanup??
- **** clean up autoidx generation
- **** use registry for server file??
- **** is scripting possible??
- **** clean up event stuff (add caching, etc)
- **** allow debug to be turned on in server??
- **** think about how his will work with threads
- **** clean up passing of data –more array stuff, by reference things?
- **** (format t " :[a ;b]" t) hangs the xlsclient read loop somehow.
- **** saving server pointers in workspaces will probably not be valid!!!
- **** handle by invalidating server pointers in com shutdown??

References

[1] Kraig Brockschmidt. *Inside OLE*. Microsoft Press, 1995.

[2] Guy Eddon and Henry Eddon. *Inside COM+: Base Services*. Microsoft Press, 1999.

[3] Hesham El-Rewini and Ted G. Lewis. *Distributed and Parallel Computing*. Manning, 1998.

[4] Sigbjorn Finne, Daan Leijen, Erik Meijer, and Simon Peyton Jones. Calling hell from heaven and heaven from hell. In *Proceedings of the ACM SIGPLAN International Conference on Functional Programming*, volume 34 of *ACM SIGPLAN Notices*, pages 114–125. ACM, September 1999.

[5] Paul Mc Fedries. *VBA for Microsoft Office 2000*. SAMS Publishing, 1999.

[6] Randal L. Schwartz, Erik Olson, and Tom Christiansen. *Learning Perl on Win32 Systems*. O'Reilly & Associates, 1997.

[7] Ellen Siever, Stephen Spainhour, and Nathan Patwardhan. *PERL in a Nutshell*. O'Reilly & Associates, 1999.

Indices

Chunks

Identifiers